ENCODING THE CLOCK CYCLE SEMANTICS OF BSV IN PVS

AN ENCODING OF THE CLOCK CYCLE SEMANTICS OF BLUESPEC SYSTEMVERILOG IN PVS

By NICHOLAS CLIFFORD CHARLES MOORE B.Eng.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy

McMaster University DOCTOR OF PHILOSOPHY (2022) Hamilton, Ontario (Software Engineering)

TITLE: An Encoding of the Clock Cycle Semantics of Bluespec SystemVerilog in PVS AUTHOR: Nicholas Clifford Charles Moore, B.Eng. (McMaster University) SUPERVISOR: M. Lawford NUMBER OF PAGES: xiv, 229

ABSTRACT

The invention of Hardware Description Languages has given hardware designers access to powerful methods of abstraction and organization, previously only available to software developers.

A high-powered means of examining properties such as reliability, correctness and safety is the creation of formal, mathematical proofs of correctness. One approach to this is the modelling of the artifact in the logic of some deductive system, such as the higher order logic of the Prototype Verification System (PVS). The ambition of this work is to demonstrate a mechanism by which a class of hardware descriptions may be used to generate such models automatically. We further demonstrate the utility of said models, using them to demonstrate non-trivial correctness properties. We also present a method of generating hardware descriptions, logical models, and proofs from a class of tabular specifications.

The language on which this method operates is Bluespec SystemVerilog (BSV), a high-level hardware description language notable for its elegant semantics. The target platform of our translation is the Prototype Verification System (PVS), which features a highly automatic theorem-proving system. The translation algorithm is discussed at length, including the reconciliation of BSV's action-oriented semantic and the Kripke semantics employed by our chosen model in PVS.

Five case studies demonstrate our methodology. In studies one and two, function blocks of the International Electrotechnical Commission (IEC) 61131-3 Annex F library are verified against tabular specifications, or generated from the same. The remaining case studies are based on the Shakti RISC-V implementation of the RapidIO subsystem. Our final case study demonstrates progress towards the verification of highly abstract and complex properties over the entire translatable subset of the RapidIO library.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ASIC Application Specific Integrated Circuit. 9

BAPIP Bluespec And PVS Interlanguage Processor. 3, 6, 7, 11, 22, 25, 29, 31, 34, 44, 62, 83, 90, 93, 98, 109, 111, 117, 131, 132, 136, 139, 147, 152, 154, 157, 159, 160, 170, 171, 178, 179, 181, 182, 184, 187–191, 193–196, 212

BSV Bluespec SystemVerilog. 3–12, 15–23, 31, 33, 34, 41, 42, 44–46, 48–50, 53, 57, 62, 63, 66–68, 70, 71, 73, 74, 77, 79, 81–86, 88–91, 93–102, 105–107, 109, 112, 114–117, 120, 121, 123, 124, 126–128, 130, 131, 136, 138, 139, 142, 146, 147, 150, 152, 154, 157, 175, 179–182, 184, 185, 187, 188, 190–194

CISC Complex Instruction Set Computer. 32, 137

CPU Central Processing Unit. 32, 137

DSL Domain Specific Language. 184

EBNF Extended Backus Naur Form. 17, 34, 35, 41

FIFO First-In-First-Out Buffer. 190

FPGA Field Programmable Gate Array. 9, 12

GHC the Glorious Glasgow Haskell Compiler. 192

HDL Hardware Description Language. 8, 20

HOL Higher Order Logic. 4

IEC International Electrotechnical Commission. iii, 16, 131

ISA Instruction Set Architecture. 32, 33, 137, 138

PLC Programmable Logic Controller. 11, 12, 140

PVS Prototype Verification System. 3–13, 15, 16, 23–25, 27–30, 34, 39, 41, 44, 48, 49, 71, 82–86, 88–91, 93, 118, 120, 121, 125, 126, 128–130, 134, 139, 144, 147, 152, 157–160, 162, 166, 167, 181–183, 185, 187–191, 193, 194, 212

RISC Reduced Instruction Set Computer. 32, 137

SBCL Steel Bank Common Lisp. 158, 191

SBV SMT-based verification in Haskell. 106, 183

SMT Satisfiability Modulo Theory. 4, 106, 107

TSP PVS Tabular Specification file format. 90, 91, 188

UML Unified Modelling Language. 11

VHDL VHSIC Hardware Description Language. 4, 12, 13, 65, 181, 182

1. INTRODUCTION

In this chapter, we will provide motivation for the work in §1.1, followed by an overview of the work in §1.2, an overview of the contributions of this work to the state of the art in §1.3, and finally, we will discuss the outline of the thesis.

1.1 Motivation

As hardware design complexity has increased with the lifting of constraints related to design abstraction and prototyping, so too has the effort required to verify hardware designs. Formal methodologies have been adopted in a wide variety of industrial applications since the 1990s (Fitzgerald et al., 2013). There are many disparate formal techniques employed in industry (model checking, formal proof, and code generation, for example) across a broad domain of industries (Transport, Telecom, Nuclear, etc.), fulfilling an equally broad domain of applications (Controls Engineering, Distributed Computing, and Hardware Design, to name a few). Fitzgerald et al. indicate an overall positive response among the applications surveyed in (Fitzgerald et al., 2013), supporting the investigation of further applications of formal methods, and formal methods tool support. Fitzgerald et al. surveyed 62 industrial applications of formal methods, including projects within the transport, finance, defence, telecommunications, and office administration sectors. In particular, those surveyed expressed a desire for simplified and integrated toolchains for the deployment of formal methods, with a greater degree of usability, and reduced operational time. The purpose of the project presented herein is to address these two concerns, within the chosen domain of application.

However, despite the increasing use of formal methods, software testing is still the primary means of software and hardware verification. The testing process roughly consists of providing strategically selected inputs to the module under test, and the comparison of the produced results to expected results. While widespread and eminently practical, it suffers from a fundamental, mathematical problem. Exhaustive testing is practically impossible for complex systems. That is to say, in order to determine whether a module is correct for every possible input, one would need to test every single input. Further, for modules which hold state, one would need to test every possible input in every possible sequence of inputs. Fortunately, in most cases, not every input needs to be tested. While all tests provide information on a system's correctness or performance, large sets of unique test cases may reveal the same information about the system under test, and exhaustively testing all such cases is a poor use of resources. In addition, determining a set of tests to reveal the maximum amount of information about the correctness of a system is nontrivial, and normally requires at least some knowledge of the internal structure of the system, to ensure that tests fully cover all behaviours of the system under test. When heuristics are employed to reduce the number of test cases to a manageable size, there can be no guarantee that meaningful test cases have not been dropped by the heuristic. Essentially, it is always possible, however unlikely, for a discounted test case to expose a bug hitherto unknown, as unpredictable behaviour is an emergent property of sufficiently complex systems (Lorenz, 1995). As a demonstration of the impracticality of exhaustive testing, let us imagine we have a module that takes as input 8 bytes of data. This results in 1.84×10^{19} possible concrete input values. If we generously assume we can test one million cases per second, it would take over five hundred thousand years to exhaustively test every possible arrangement of memory. Not only is this a very long time for a very small program, but each bit we add to the input doubles this value. This is commonly known as the state explosion problem. Although software testing can provide high levels of confidence, there is always the spectre of the untested catastrophic failure case that, however remote the possibility, cannot be discounted completely.

However, formal methods cannot provide the mythical 100% guarantee of correctness (as some advocates claim, according to (Hall, 1990)). While closing the gap between formal models and the programs they represent remains an active area of research, even with respect to Bluespec SystemVerilog (Choi et al., 2017), there is a problem inherent to modelling itself. That is, anything proven for a model is only proven for the model. For example, classical mechanics are a series of mathematical models of the motion of objects. It should not be claimed that these models predict with 100% accuracy the motion of all objects in the universe. Many more models have been created over the course of the development of physics as a science, which supplement our understanding of motion, to bring that understanding more in line with the behaviour of objects. Formal methods do a similar thing with hardware and software. The key question is not, is our formal model 100% accurate to the behaviour of the physical system, but, is our model accurate enough for the conclusions we draw to be largely applicable to the underlying physical system.

In this work, we make the argument that the hardware model in the Bluespec language is sufficient to demonstrate useful properties on examples that are large enough to be of practical value, as demonstrated by our case studies. We also delineate where our current understanding of the physical behaviour of Bluespec derived hardware breaks down, and where more refinement would be necessary to demonstrate more advanced properties.

In general, mathematical proof generally begins with the modelling of the system under examination as a set of axioms (such as those of set theory and propositional logic), predicates and propositions in first order or higher order logic. The property the user is interested in is also expressed as a logical theorem, and the user then attempts to demonstrate, by means of inductive and/or deductive reasoning that the conclusion may be reached from the given premises. The production of mathematical proof requires a high degree of mathematical skill and training, and a great deal of effort, which significantly contributes to the lack of proliferation of formal mathematical proof employed in private-sector software and hardware development.

Modern formal verification is performed using software tools which reduce the effort required to produce proofs of correctness. The purpose of this thesis is to present one such tool, developed by the author, which reduces the verification effort required to demonstrate the validity of hardware designs.

The intention of the work herein presented is to demonstrate a novel means of verifying hardware components via formal mathematical proof. These hardware components must first be expressed in the language Bluespec SystemVerilog (BSV) (Nikhil, 2004). Using the purpose-built program BAPIP, hardware descriptions encoded in Bluespec SystemVerilog (BSV) are translated by means of logical model extraction and static analysis into PVS (Owre et al., 1992). Once encoded in the higher order logic of PVS, verification theorems represented as logic sequents may be proven using the PVS theorem prover.

This work documents the author's progress towards the problem of automatic formal verification of hardware descriptions written in Bluespec System Verilog (BSV). BSV was selected as a language for several reasons. First, its design philosophy and origin as a library of the functional language Haskell (Hudak and Fasel, 1992) set it apart as one of only a handful of hardware languages based in the functional paradigm (Chen, 2012). The guarantees made about the atomicity of transactions, and a state-machine approach enforced at a syntactic level lend BSV to formal modelling and reasoning much more easily than the majority market-share languages, such as VHSIC Hardware Description Language (VHDL) and Verilog.

PVS was selected as a modelling environment due to the power and reliability of its automatic deduction strategies, and because PVS was used by our predecessors in this field, Richards and Lester Richards and Lester (2011), who published a methodology for the embedding of BSV descriptions in PVS which largely preserved the syntactic structure of the original BSV. The original embedding by Richards achieved a high degree of similarity between parts of his BSV and PVS code through the use of monads in PVS Richards (2011b). While the use of monads necessitates the use of a theorem prover with higher-order logic capabilities, the choice of PVS by Richards is not well motivated. Yices integration and Satisfiability Modulo Theory (SMT) solving is cited as a point in PVS's favour, but this functionality goes unused in Richards' work. Only one other automated proof tools making use of higher order logic cited by Richards and Higher Order Logic (HOL) (Gordon and Melham, 1993) and Isabella (Wenzel et al., 2008), but no reason is given for preferring PVS. In particular, Coq seems a natural choice (Bertot, 2008), and is popular among related work, but is only mentioned as a "type theory tool" (Richards, 2011b) in related work. Early versions of the work presented here sought to automate Richards' manual translation, so PVS was selected by this choice in the project's earliest phases. As the project evolved and departed from Richards' embedding, the monadic aspect was also dropped, but enough work had gone into PVS as the target language of translation that this choice was upheld. The project only began to push on what is possible in PVS in its latest phases (see §6.5), at which point switching to another tool was infeasible. PVS does indeed have many qualities to recommend it, including a highly legible specification language, counterexample generation, and highly automatic deductive strategies. Further, the translator has been modularily designed, so that the translation given could in theory be modified to support several theorem proving engines, so long as something like state records and transition predicates can be expressed (see §A).

Our work's relation to that of Richards and Lester is discussed in detail in §2.3.

The remainder of this chapter provides an overview of the proof process and toolchain in §1.2, highlights the contributions made by this work in §1.3 and describes the organization of the thesis in §1.5.

1.2 Overview of Verification Methodology

Two separate but complementary verification toolchains are developed. One is a verification methodology for hardware designs which have been encoded in BSV. The other takes as input tabular expressions of hardware module requirements expressed in PVS's specification environment, and generates as output BSV hardware descriptions, while also optionally generating PVS proofs of correctness of the generated BSV descriptions.

1.2.1 BSV to PVS

The primary proof method presented herein is intended as a general-purpose mechanism for encoding BSV designs in PVS, with additional optional processes supporting the production of proofs. The proposed workflow is outlined in Figure 1.1. Solid arrows indicate manual processes. Dotted arrows are automatic processes requiring minimal human intervention, and dashed arrows indicate partially automated processes.

Minimally, a BSV description of a hardware module is necessary for translation. The translator must be invoked to generate a logical encoding in PVS. The user may then use the PVS proof system to attempt to prove any condition or predicate which can be expressed in PVS.

Optional additions to this method require the existence of a specification document from which the BSV description can be created, as shown in Figure 1.1. In addition, the formalization of the specification document into tabular expressions encoded in PVS, when appropriate, provides a set of requirements that may be used directly to verify the generated PVS encoding of the BSV source. Tabular specifications, specifically function tables, have been selected as our primary mathematical formalism for expressing requirements due to their ability to express the conditions for the total correctness of a hardware module (see §2.7), and to maintain consistency with other research within our research group (Pang et al., 2015). However, as demonstrated in §6.5, such formalization need not take the form of tabular specifications. Case studies 1 through 4 (§6.1, §6.2, §6.3, §6.4) target the

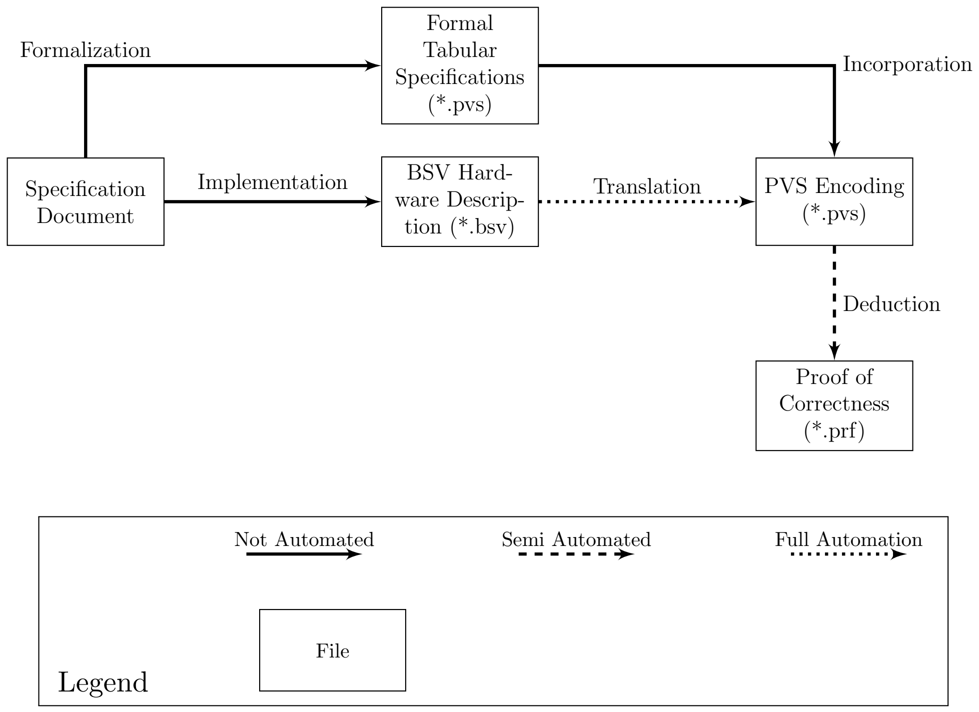


Figure 1.1: BAPIP - BSV2PVS mode toolchain

complete correctness of the hardware modules under study, so tabular specifications are appropriate, whereas case study 5 (§6.5) examines one property of one data pathway through a highly complex series of hardware modules. In the context of this thesis, complete correctness of such a system would not be a feasible goal.

1.2.2 Tabular Expressions to BSV and PVS

For certain types of specifications, we have developed a second toolchain to generate both BSV hardware descriptions, and proofs of correctness in PVS, in a single step. In order to accomplish this, certain assumptions and design decisions have been encoded in the BAPIP translation tool, which are not claimed to be optimal. This tool supported workflow is illustrated in Fig 1.2, where all processes which can be fully or partially automated have been indicated with dotted or dashed arrows respectively.

The major advantage to this technique over previous methods, such as

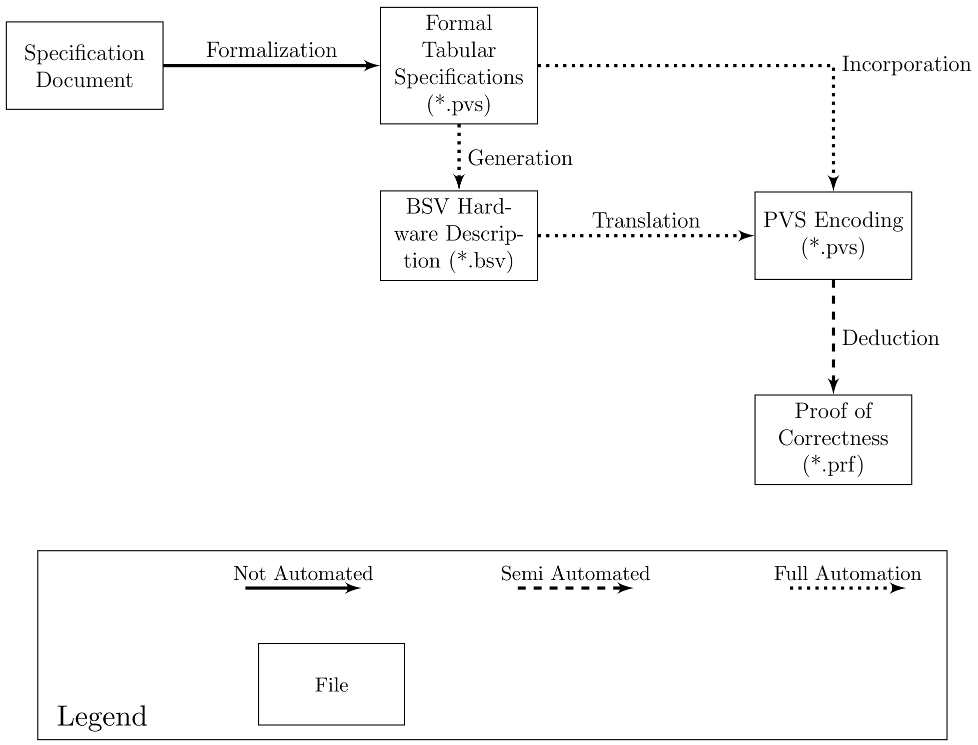


Figure 1.2: BAPIP - TSP2BSV and TSP2PVS mode toolchains

that proposed by Richards and Lester (2011), is the much greater degree of automation available. Once a formalized tabular specification has been produced in PVS, it is possible to automate all other aspects of the toolchain. This comes at the expense of a restricted set of hardware modules that may be generated, and the types of specifications which are acceptable.

The toolchain begins with specifications, which must be formalized by the user into tabular specifications in PVS. From there, BSV descriptions may be automatically generated via an invocation of BAPIP. The invocation selected may either be the tabular specification to Bluespec SystemVerilog translator, which only generates the associated hardware description, or the tabular specification to PVS proof obligation generator, which generates the Bluespec SystemVerilog hardware description, its encoding in PVS, as well as proof scripts for PVS which, when invoked by the proof engine, result in the verification of the generated BSV description against the tabular specifications originally used to generate them. For more detail, see §2.2.1.

1.3 Contributions

The work presented in this thesis extends the state of the art in the field of the tool-assisted verification of Hardware Description Language (HDL)s as follows.

- A novel translation algorithm is presented for the embedding of BSV in PVS. While some aspects of this embedding were originally based on work previously published by Richards and Lester (Richards and Lester, 2011), most of the previous work has been superseded, with only a few core concepts persisting into the current iteration. However, this work can be seen as an enhancement of the work of Richards and Lester in a few key categories.

  1. The embedding presented by Richards and Lester was a manual process, whereas we present software automating the translation.
  2. The subset of the source language over which our tool operates is much larger than the subset of BSV demonstrated using Richards and Lester's manual embedding. This allows our tool to be applied to real-world embeddings, of a type not possible in previous works.
  3. The semantics encapsulated by our methodology is a more accurate simulation of the semantics underlying BSV itself, addressing key components of the BSV semantics never addressed by the embedding presented by Richards and Lester, such as full clock cycle semantics.
  4. The embedding of BSV's whole clock-cycle semantics in a logical system and the application of formal methods to prove logical properties over these whole clock cycle semantics, is something which has never before been attempted. All other formal methods projects on Bluespec SystemVerilog stop at single-step semantics, and do not compose these single steps into full clock cycles.

  The goals of this work are that:

  - The software translation tools are useful to those wishing to perform their own verifications of hardware modules

  - Some of the algorithmic methods used to move between state and rule based systems may be informative to future translation efforts.
  - Some of the algorithmic techniques addressing the state explosion problem presented in §4.4 may be useful in addressing scalability in future similar systems.

- We also present a tool-assisted methodology for the generation of BSV hardware descriptions directly from tabular specifications expressed in PVS, as well as a method for generating proofs of correctness for these hardware descriptions automatically.

- We present full examples and practical instructions for the development of verified Bluespec SystemVerilog modules from requirements tables, with multiple pathways for varying levels of user participation. This includes :
  - practical instructions for the verification process, including how to use, construct, and integrate theorems with tabular specifications and other types of conditions for correctness.
  - five case studies demonstrating our methodology.

1.3.1 Publications

The work of this thesis has garnered recognition by the broader scientific community in the form of the following two paper publications.

- Moore, N., & Lawford, M. (2022, July). A Case Study in the Automated Translation of BSV Hardware to PVS Formal Logic with Subsequent Verification. In International Symposium on Theoretical Aspects of Software Engineering (pp. 65-72). Springer, Cham.

- Moore, N., & Lawford, M. (2017, May). Correct safety critical hardware descriptions via static analysis and theorem proving. In 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE) (pp. 58-64). IEEE.

1.4 Related Work

Given the prevalence of embedded systems in safety-critical applications, and the use of Hardware Description Languages in both Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) design, it is unsurprising that significant scientific effort has been directed toward the mathematical verification of these design languages. It should be noted that the most related work to this thesis, i.e., that of Richards and Lester (2011) and the Shakti project (Menon et al., 2017) (upon which several case studies have been based), have been discussed in detail in §2.3 and §2.4 respectively.

1.4.1 Verification Work on Bluespec Designs

A major development in the formal methods applications of Bluespec SystemVerilog is Kami (Choi et al., 2017), advertised as "a platform for high-level parametric hardware specification and its modular verification." This paper by Choi et al., was co-authored by Vijayaraghavan, who has made other major contributions to the formal methods applications of BSV (Vijayaraghavan et al., 2015), and Arvind, who was influential in the early development of the Bluespec language (Nikhil, 2008). Kami is an extension of Coq (Bertot, 2008) which duplicates a subset of BSV semantics within the Coq framework. The essential verification workflow is to generate both a specification module and an implementation module in Kami, and then to demonstrate a refinement relation via module substitutions, inlining method calls, and demonstrating rule and method correspondences (Vijayaraghavan et al., 2015).

Similarly to the work presented here, and the work of Richards and Lester (2011), Kami models Bluespec actions as a state transition system. Similarly to Richards and Lester, and dissimilarly to the work presented here, that formal work stops short of whole clock cycle semantics. The expected workflow in Kami starts with composing the hardware design inside of Kami (and therefore Coq), where individual rules can have their behaviour verified. From there, Kami code is **de-sugared** into Bluespec, and the Bluespec compiler itself is used to compose the whole clock-cycle semantics. As such, many of the issues with the Richards and Lester embedding in PVS recur in Kami. While it is certainly useful to be able to verify the action of a single rule, scheduling behaviour is not sufficiently straight-forward to not be worth the effort of closer examination. Under one-rule-at-a-time semantics, if you want some property to hold for any combination of actions, it must be proved for each individual action. While this is useful for properties such as safety conditions, there are many behaviours which are interesting to prove which are not provable for every action in a module. For example, if two actions are in conflict, and write different values to a register, these two actions would not both prove the register was written with either of the particular values written. This is also not a completely vacuous example, as this conflict could be resolved via means other than guard exclusivity, and be a perfectly valid design. Kami also explicitly does not a address "constructs that violate one-rule-at-a-time semantics [...] namely "wires," whose behaviour depends on the schedule" (Choi et al., 2017). It is only possible to simulate wires in the context of whole clock cycle semantics, so this is an understandable omission in Kami, but not one that is shared by BAPIP.

Kami defines a syntax for an acceptable subset of BSV, and semantics for making "judgments" on this syntax. These judgements are stylistically similar to operational semantics. In Kami, a specification model is written in a combination of Kami syntax and Gallina, the specification language of Coq, and an implementation is written in Kami, without the Gallina component. Choi then attempts to prove the existence of a multi-step judgment relation between the specification and the implementation. This approach sharply contrasts our approach to the encoding of specifications. Our approach is to expose BSV semantics to a mechanized logical system of reasoning in which a module's requirements may be directly expressed, as nearly as possible to their original or natural expression in formal logic. While the utility of Kami to a Bluespec designer is undeniable, our approach is intended for a more general audience. Aside from a recent case study (Uma et al., 2022), there seems to not be much recent published work on Kami.

While the generation of code for embedded systems is a popular topic of model based engineering (Rashid et al., 2015), these techniques most often start from modelling languages such as Unified Modelling Language (UML) or similar, which may lack the interdisciplinary legibility of requirements. In the case of Durand and Bonato (2012), finite state machines encoded in UML are used to generate BSV templates. It is intended that users populate states with behaviour manually, as only the state transition mechanism is converted. While our technique operates on a less abstract model, the BSV files generated by BAPIP have no need for manual intervention in TSP2PVS mode, which accepts tabular specifications encoded in PVS and produces both BSV descriptions from said specifications and translates this generated hardware module to PVS. S. Durand has no publications past the original 2012 paper. Though Durand and Bonato (2012) is cited by a number of papers, none seems to be a direct descendant of the orignal work.

The work presented here was originally intended as a parallel verification effort to the work of Pang et al. (2015), in an effort to improve the degree of assurance provided by a pre-verified library of Programmable Logic Controller (PLC) function blocks extracted from (IEC, 2013). Pang's work has since seen a great deal of development by Newell et al. (2018). It is projects such as this that our work is intended to be useful for.

Another hardware verification strategy targeting Coq is Featherweight Synthesis (Fe-Si) (Braibant and Chlipala, 2013). Similarly to BAPIP, this tool translates a subset of Bluespec SystemVerilog to a proof environment, in this case Coq. This work follows more in the line of Richards and Lester (2011), by modelling state transitions monadically, rather than using records such as BAPIP. Similarly to Blech and Biha (2013), verification is dependant on the advanced skills required to operate in a dependently typed proof environment. Fe-Si is presented as a proof-of-concept, and expansion to a more practical subset of BSV is cited as a goal of the project, in contrast to BAPIP, which verifies an industrial-strength subset of BSV. Additionally, and similarly to Kami, Fe-Si avoids the problems of action conflicts and wire semantics by leaning on the fallacious expansion of one-rule-at-a-time semantics to address the issue of composition, and excluding wires from consideration. It seems Fe-Si didn't go anywhere, but was co-authored by A. Chlipala, who has co-authored a number of other Bluespec-related papers. It is possible that Fe-Si was influential in the eventual design of Koika (Bourgeat et al., 2020).

Tools also limited in language scope are the Theosim translation tool (Morin-Allory and Borrione, 2006), and to a lesser degree its subsequent re-implementation VSYML (Ouchet et al., 2009). Both of these model the simulation semantics of VHDL in theorem proving environments. Theosim targets PVS, and VSYML has been designed to accommodate add-on backends for a number of proof systems. Both require simulation of the hardware description in order to generate logical statements, whereas our approach derives logical sequents primarily from the manipulation of abstract syntax trees. Hardware simulation can be quite time consuming, but is a fundamental testing mechanism in hardware development. Because they're starting with VHDL, a large language which lacks the semantic elegance of Bluespec, a simulation approach is quite understandable. Neither project has had any recent citations.

Newell et al. have published a tool-supported method for translating Tabular Expressions and FBDs into PVS for the purposes of verification (Newell et al., 2016). This method is presented within the context of verifying the safety shutdown system of the Darlington Nuclear Power Generating Station. While powerful and useful, this approach does not leverage the power, efficiency and flexibility of FPGA based control systems. This paper is coauthored by Linna Pang, whose PhD work on PLC function block verification was a direct antecedant to this thesis.

Bidmeshki et al. present a translator for converting Verilog code to Coq (Bidmeshki and Makris, 2015), for the purposes of detecting hardware "Trojans," or, malicious code designed to infiltrate designs and violate data flow policies. The emphasis here is proof-carrying hardware code, and the translator goes so far as to automatically construct theorems to verify the enforcement of data flow policies. This work demonstrates a considerable effort towards usability, which is always welcome in the formal methods community, but unfortunately the application domain is too narrow for this to be a good general purpose tool. Burlyaev et al. demonstrate a very similar strategy, but focusing on fault-tolerance rather than hardware Trojans (Burlyaev, 2015).

Vijayaraghavin et al. present a prototype embedding of Bluespec in Coq (Vijayaraghavan et al., 2015), similarly to the Richards and Lester paper on which our embedding in PVS is based. They present a far more complex example than Richards and Lester, a multi-core shared memory system, and indicate that they are working on a software tool for automating this translation process. This, presumably, either influenced or resulted in Kami (Choi et al., 2017).

Saeed et al. present a VHDL to HOL4 translation called V-Holt (Saeed et al., 2012), in which users specify the properties they want checked in a Java-like language. In addition to the translation, the V-HOLT tool also produces proof strategies for the specified properties, and returns to the user a simple pass/fail output. This project is a paragon of user-friendliness, but there are few pitfalls of such a technique. If V-HOLT fails to prove something, that does not mean that it is unprovable, as no algorithm can completely replace the insight and intuition of a human operator. Further, specifying the desired properties in this Java-like language must be easier than specifying them in HOL4 in order to justify such an approach. This work has had no recent citations.

Ostroumov (Ostroumov et al., 2013; Ostroumov and Waldén, 2015) demonstrate the generation of VHDL from EventB models. The resulting VHDL is claimed to be correct by construction. This is the reverse of our approach, which generates models from hardware descriptions. Later derivative work focuses on languages such as Java (Catano, Néstor and Rivera, Víctor, 2016) and C++ (Bonfanti et al., 2020).

1.4.2 Other Projects of Interest

Koika, by Bourgeat et al. (2020) is a high profile project to improve Bluespec semantics. As with Kami (Choi et al., 2017), Koika is a product of the same MIT group who originated Bluespec. Koika is primarily an attempt to address the gap between the one-rule-at-a-time semantics of the Bluespec language and the whole-clock-cycle semantics. Their paper, "The essence of Bluespec: a core language for rule-based hardware design" presents many of the same scheduling difficulties as are presented in this very thesis. It is claimed that Bluespec and Koika are both implementations of the one-rule-at-a-time semantics. One interesting point of comparison between Bourgeat et al. (2020) and the Bluespec documentation is how it is claimed the semantic works in the absence of a priority ordering of rules. The Bluespec document claims firmly that an "arbitrary but deterministic" decision is made by the scheduler circuit generated by the Bluespec compiler in ambiguous cases. In our work, we seek to eliminate this by not permitting any design to translate wherein such ambiguities lie. The claim of Bourgeat et al. (2020) is that this decision is fully "non-deterministic". Whether this is a misunderstanding on Bourgeat's part, a shift in policy on the part of the Bluespec research group, or whether this was the hidden truth the whole time remains to be seen. The manner in which Koika solves this issue is by requiring the user to manually schedule rules. In the opinion of the author, this looks a lot like taking the descending urgency pragma from Bluespec and promoting it to a first-class member of the language.

Koika also seeks to address the problem of inter-action communication in a novel way. Bluespec (and most hardware languages) solve this problem by designating named "Wires". In Bluespec, an action may receive information from another action, so long as that action is transmitted via a wire, and that wire is written to before it is read from. Koika's scheme involves labelling registers(!) with read-write priorities. This relies on an interpretation of the one-rule-at-a-time semantics where the results of each rule's calculation are visible to the next rule to fire. Effectively, this means *every register is also a wire*. In Koika, these semantics are combined, so that read-write priority is set at the level of the individual registers.

Koika is attempting to address the observed behaviour of Bluespec designers. It is noted that designers are often using somewhat arcane methods in an effort to make Bluespec designs more parallel, and therefore faster. The design philosophy of Koika seeks to eliminate the need for this, by exposing more of the scheduler to the designer.

Koika has been provided formal semantics in Coq, which have been used to produce a verified compiler, and most interestingly, a formal proof that Koika semantics observe the one-rule-at-a-time semantics. It is an interesting attempt to have one's cake and eat it too. Each rule can be treated as if it is the only one executing in a clock cycle, yet through use of wire-register hybrid constructs (termed "Ephemeral History Registers") we can execute many such actions in one clock cycle. Comparisons of algorithm processing times between Bluespec and Kioka indicate Bluespec designs are slightly more efficient citepBourgeat2020.

The catch is that Koika is, as yet, not a mature language. According to github commit logs (at time of writing) koi (2022), active development ceased in early March of 2022. Certain things one would expect are still missing, including a detailed language reference manual. Further, claims are made that there is still work to be done in optimizing the post-compilation RTL designs koi (2022). Aside from these points, Koika also takes Bluespec syntax, which has a sort of Fortran/C feel, and giving it a much more functional feel, perhaps more closely resembling the original Bluespec Haskell libraries.

One project of interest is Π-ware, a project by Pizani Flor (2014). This project is to Agda as Bluespec was to Haskell, aiming at proof-carrying hardware code with dependent types in a functional environment. Hardware modules written in such a system would theoretically require no further verification effort, but would require considerable effort to compose. Π-ware seems to have gained some traction, having had an official release citep-pizani2018pi, and several recent citations (Lööw, 2021; Lööw, Andreas and Kumar, Ramana and Tan, Yong Kiam and Myreen, Magnus O and Norrish, Michael and Abrahamsson, Oskar and Fox, Anthony, 2019; Lööw, Andreas and Myreen, Magnus O, 2019).

Another interesting project is presented by Brandt et al. (2010), which translates concurrent, action-oriented specifications (similar to Bluespec) into scheduled actions, explicitly exposing the schedule which some languages (like Bluespec) have historically put a lot of effort into hiding.

Daylight and Shukla (2009) presents an overview of four specification languages, including Bluespec. Bluespec rates highly for maximizing throughput, but rates poorly in local reasoning and adaptability, though it should be noted that this is an evaluation of Bluespec as a specification language, not a hardware description language. Stappers et al. (2010) extends this study to mCRL2, a specification language for describing concurrent discrete event systems.

1.5 Organization of the Thesis

Chapter 2 presents preliminary background information, including brief outlines of the source and target languages of the translation process (§2.1 & 2.2), as well as other topics of relevance.

In Chapter 3 we present the grammar of the accepted subset of both the source language (§3.1), and abstract syntaxes for both BSV and PVS, as expressed as types in Haskell (§3.1 & §3.2). The generation of PVS from abstract syntax is discussed in the chapter discussing BSV's action arbitration semantic (§4). §3.3 discusses why providing a full syntax for the resultant PVS specifications is beyond the scope of this work, and in §3.4 the formal grammar used to parse tabular specifications is discussed.

The core algorithm of the BSV to PVS translation is presented in Chapter 4, where we explain the process for transforming the action-centric BSV representation into the state-centric PVS representation. The action arbitration semantic is presented in detail with a running example in §4.2, and steps taken towards the optimization of the above algorithm are presented in §4.3, with a focus on improving the scalability of the algorithm.

Chapter 5 concerns the production of proofs of correctness from translator results. We address the construction and use of theorems demonstrating both functional requirements (§2.2.1) and consistency (§5.3).

Chapter A discusses the translation software itself, including its architecture (§A.1), restrictions (§A.3), and instructions for operation (§A.4).

In Chapter 6, we present several case studies. In §6.1, we present a verification case study over a small example from the IEC 61131-3 function block library (IEC, 2013). This case study is based on one originally published by the author at the 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE) (Moore and Lawford, 2017). We here present an updated version of the same verification, which has been streamlined and simplified to make use of later expansions of the translation software. Second, we present in §6.2 the verification of a similar hardware module, making use of the translator's ability to generate BSV descriptions and PVS proofs of those descriptions from tabular specifications encoded in PVS. Third, in §6.3 we verify a more complex function from the Shakti RISC-V processor project's RapidIO interconnect fabric. Finally, in §6.5, we show progress towards the partial verification of the interconnect fabric. Specifically, the handling of transaction ID numbers in packet processing.

We conclude in Chapter 7, summarize our results, and discuss future work.

2. PRELIMINARIES

This chapter is presented for the benefit of those who may be unfamiliar with the subject area of this thesis. We will begin by discussing Bluespec SystemVerilog (BSV) in § 2.1, a hardware description language with semantics based in the functional paradigm. The Prototype Verification System (PVS) will then be discussed in § 2.2, including its structure and the reason for its selection. Next we will address the groundwork for our approach from (Richards and Lester, 2011), as well as our departures from this model, in § 2.3. We will also provide a brief discussion of the RISC-V microprocessor standard, the RapidIO interface, and the Shakti processor project (Gala et al., 2016) in § 2.4. In § 2.5 we review the Extended Backus Naur Form (EBNF) (Extended Backus-Naur Form) notation that will be used in this thesis. A primer on the Haskell type system is provided in §2.6. Finally, an overview of tabular specifications, as well as some properties of note, is presented in §2.7.

2.1 Bluespec SystemVerilog

Bluespec SystemVerilog (BSV) is a high-level hardware description language (HDL) influenced by the functional programming paradigm (Nikhil, 2004). It is designed as an alternative to the so-called "ad-hoc," lower-level languages, exemplified by Verilog, SystemVerilog and VHDL. BSV's main claim is that it makes the benefits of programming in abstract languages available to codified hardware design (Nikhil, 2004).

Bluespec originated as a library of the Haskell programming language (Hudak and Fasel, 1992; Nikhil, 2004), and has since developed into a fully fledged hardware description language in its own right. Bluespec Inc., the company which produces and distributes Bluespec, was founded in 2003 by Dr. Arvind of MIT. Despite claims made by Bluespec Inc. of the increased speed and reliability of BSV versus conventional languages, Bluespec was primarily a language of academic interest, until the Shakti RISC-V project

(Gala et al., 2016), at which point Bluespec Inc. pivoted to become a provider of RISC-V technologies (Bluespec Inc., 2019).

2.1.1 The Bluespec Language at a Glance

In order to discuss the detailed semantics of the Bluespec language, a basic introduction to the language is necessary. We will illustrate this using the example of a stoplight.

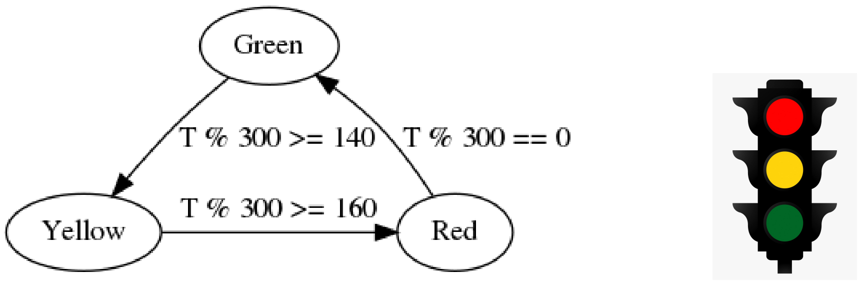


Figure 2.1: State Transition Diagram for a Stoplight

In this very simple system, different lamps are lit based on the value of the input signal `T`, which is the time in seconds since the beginning of the day. The system cycles through the green, yellow and red lamps being lit every 300 seconds, with the green and red lamps being lit for 140 seconds out of each cycle, and the yellow lamp being lit 20 seconds of each cycle.

This system could be expressed as a BSV hardware description in the following manner.

BSV

```
package TrafficLight;

  typedef enum {Green, Yellow, Red} Colour;

  interface TrafficLight;
    method Action setTime (Int#(32) t);
    method Bool getGreen ();
    method Bool getYellow ();
    method Bool getRed ();
  endinterface
  module mkTrafficLight (TrafficLight);
    Reg#(Colour) lampState <− mkReg(Red);
    Reg#(Int#(32)) T <− mkReg(0);

    rule goYellow (lampState == Green && (T % 300) >= 140);
      lampState <= Yellow;
    endrule

    rule goRed (lampState == Yellow && (T % 300) >= 160);
      lampState <= Red;
    endrule

    rule goGreen (lampState == Red && (T % 300) == 0);
      lampState <= Green;
    endrule

    method Action setTime(t);
      T <= t;
    endmethod

    method Bool getGreen;
      return lampState == Green;
    endmethod

    method Bool getYellow;
      return lampState == Yellow;
    endmethod

    method Bool getRed;
      return lampState == Red;
    endmethod
  endmodule
endpackage
```

END BSV

The foregoing Bluespec description is nearly a direct implementation of the state transition system described in Figure 2.1. The arrows of the diagram, which direct the order and timing of the lamps, are directly expressed as `rule` constructs in BSV. Each rule is "guarded" by a Boolean expression, and transforms the values of state elements (registers). In the same way that an arrow in Figure 2.1 provides a transition between different lamp states, the rules in BSV provide transitions between different states in register memory.

The methods in this description act in the same manner as getter and setter methods in object oriented programming. The `setTime` method sets the internal `T` register with a time value, derived from an external hardware component. Similarly, signals to the traffic light lamps are extracted from this hardware module by means of the three methods "getGreen", "getYellow" and "getRed". Each of these methods will send a high or low voltage signal, depending on the results of the combinational circuits interpreted from the return expressions.

One item of note, and a dissimilarity from conventional hardware description semantics, is that hardware module inputs and outputs are accessed *atomically*. That is, rather than a wire being connected in an "always on" fashion, data is transferred between BSV hardware modules only when specifically requested by method invocation. In this manner, BSV modules follow software language semantics more so than hardware language semantics.

BSV modules and their interfaces are declared separately, and a module must specify which of the available interfaces it uses. This allows interfaces to be abstracted, and shared across multiple Bluespec modules. A Bluespec package may contain an arbitrary number of modules.

2.1.2 A More Detailed Semantic Overview of a Bluespec Package

From the top-level, Bluespec follows roughly the same organizational scheme as SystemVerilog; whereas packages are collections of module declarations. Like SystemVerilog, and functional languages generally, Bluespec is a declarative language. Declaration of modules are distinct from instantiations, and, as with SystemVerilog, a module may instantiate any number of submodules. This gives rise to a hierarchical organization of modules, wherein each instantiated module has exactly one parent which instantiates it, except for the root-level module, of which there may be only one. A module may have any number of children which it instantiates.

The manner of passing data between modules differs significantly from SystemVerilog. Whereas in SystemVerilog a module's interface provides a number of identifiers corresponding to input and output wires which may be used more or less freely throughout the module, a Bluespec module's interface provides the user with a set of access methods, which function similarly to how one might expect an Object Oriented language to behave. Various claims are made of Bluespec's methods, and the improvement they constitute over the regular HDL model. To fully appreciate the difference, however, it is first necessary to discuss the internal semantics of Bluespec Modules.

Bluespec modules have two elementary components: submodule instantiations and guarded actions. In §2.1.3 we will discuss the ease with which this system may be modelled logically as a Kripke structure.

Submodule instantiation refers not only to modules defined by the user, but also Registers, FIFOs, and other organizations of memory. Some syntactic sugar is applied for ease of use (particularly in the case of registers), and this partially obfuscates the fact that they are invoked and used in precisely the same manner as any other user-defined submodule. Even in the case of user-defined submodules, unless the submodule is purely combinational, it will itself contain instantiations of memory-holding elements. Therefore it can be considered to hold state, albeit in a somewhat more complex structure which also specifies the manner of interaction with those memory-holding elements. As such, a user-defined Bluespec module allocates and accesses memory exclusively through the use of instantiated submodules, which become synonymous with the concept of state.

Guarded actions in a BSV module manipulate state. A guarded action is somewhat predictably composed of a guard and an action, where a guard is a Boolean expression that limits the conditions under which the action may be executed. In order for an action to execute in a particular clock cycle, the corresponding guard expression must evaluate to "True", when given the state of any invoked memory elements for their values at the beginning of the clock cycle under examination. An action consists of one or more statements, normally invoking submodule methods, which describe changes in the module state over one clock cycle. The statements within an action operate concurrently, not sequentially, and therefore every statement in an action must modify a distinct memory element. There are three types of guarded action in a Bluespec module: Rules, Methods and Actions.

To begin, Actions are blocks of statements with no guard that may be invoked as statements themselves. One practical use of Action blocks is to encapsulate commonly used statements, such as a reset procedure. This allows these statements to be collectively invoked by action name, rather than having code snippets copied and pasted haphazardly throughout one's design. In short, separately declared action blocks are abstraction tools for improving code quality, but do not constitute distinct semantic entities.

A "Rule" is a guarded action which is always available, but not always executed. One way of thinking about them is as "passive actions," which require no invocation from a parent module in order to execute. That being said, calculating which rules will indeed execute on any given clock cycle is non-trivial. Only "active" rules, which have had successful guard checks, may execute. But only rules which succeed on action arbitration will actually execute, or "fire." The action arbitration semantic is a complex issue, addressed in §4.

Methods, as mentioned previously, are the means by which a parent module interacts with its children. They may be thought of as "active actions," which must be invoked by a parent module to be available. When invoked, methods must still pass a guard check, for example, a design may decide to use a guard to restrict access to a module's output data while that data is invalid. Methods take priority over rules during action arbitration, and will automatically succeed against them during conflict resolution.

In BSV, guarded actions are atomic, and will execute completely or not at all. While it is possible to write a BSV description that produces race conditions with respect to memory access, this will produce compiler warnings which, due to the sparse nature of warning and error messages when developing BSV descriptions, are unlikely to go unnoticed. For the purposes of the BAPIP translation tool and the work presented herein, BSV descriptions are required to resolve any ambiguities of this nature via pragmas, which may be used to pass instructions to the action arbitration mechanism.

A more precise description of the syntactic construction of Bluespec SystemVerilog will be presented in §3.

2.1.3 Logical Abstraction of Bluespec Modules

For the purposes of logically modelling BSV descriptions, we may collectively consider memory-holding submodules to have some particular state on any given clock cycle, and we may consider the actions, which govern all of the interactions between submodules, to be a system of transitioning between different states. As such, BSV descriptions may be modelled as finite state machines, and may be described logically as Kripke structures (Bowen, 1979), as proposed by Richards and Lester (2011),

$$K = (S, s_0, T, L), \tag{2.1}$$

where S is the set of program states, $s_0 \in S$ is the initial state, T is a left-total transition relation on S, i.e.,

$$T \subseteq S \times S \tag{2.2}$$

$$\forall s \in S \bullet \ (\exists s' \in S \bullet \ (s, s') \in T) \tag{2.3}$$

and L is a labelling function. In BSV it is mandatory to initialize declared state elements, and this declared arrangement of memory comprises the initial state s_0. Rules and methods, along with BSV's internal semantics of rule arbitration, create the set of valid state transitions. This formal logical approximation of BSV descriptions is of central importance to the PVS embedding presented by Richards and Lester (2011) and forms the logical framework for our own translation.

2.2 Prototype Verification System

The Prototype Verification System (PVS) is an interactive specification and proof environment, providing both a high degree of mechanization and the expressive power of higher order logic (Owre et al., 1992). This free and open source tool, developed by SRI International, consists of an Emacs-like environment, where logical expressions and theorems are encoded by the user, and an interactive proof environment, in which the user may apply proof tactics to prove theorems. The proof environment automatically generates sub-goals and counter-examples, exhibits high mechanicity, and produces highly legible proofs. Users may opt for step-by-step manual control, or invoke high-level proof strategies. Herein, "PVS" will signify either the proof environment itself or the associated specification language contextually. In the past, PVS has been used to successfully verify safety critical embedded systems, such as the AAMP5 avionics microprocessor (Miller and Srivas, 1995), and shutdown systems for the Darlington nuclear power plant (Wassyng et al., 2011). This combination of mechanicity, legibility, and a logic expressive enough to accommodate our extracted BSV program models, recommends PVS to this project. Use of PVS to verify our BSV modules will be presented in §4.

The previous work of Pang et al. (2015) makes use of PVS for the specification of tabular expression for the IEC61131-3 function block library. The work of Richards and Lester (2011) make PVS the target language of an embedding procedure for BSV. While there are many languages which would have been suitable target languages for the translation algorithm specified herein, in the interest of integration with directly relevant research PVS was selected as the target of our translation algorithm.

2.2.1 Constructing Theorems in PVS

To prove some property of the logic generated by BAPIP, we must postulate this property in a manner which PVS can parse: a `theorem`. According to the Collins English Dictionary, in logic, a theorem is "a statement or formula that can be deduced from the axioms of a formal system by means of its rules of inference" (Forsyth, 2014). In PVS, we may use the `theorem` keyword to construct hypotheses. That is, sequents that may or may not represent theorems. Since a sequent is only a theorem in mathematical logic if the theorem is demonstrable from axioms by means of a deductive system, until such a sequent has been demonstrated to be true (using PVS's theorem proving system), it is not yet a theorem. Usage of the word "theorem" in PVS seems more to follow the mathematical definition of the word, rather than the logical definition. According to the American Heritage dictionary, a theorem in mathematics is "A proposition that has been or *is to be* proved on the basis of explicit assumptions" (Kleinedler, 2016) (my emphasis). Given that, in PVS, we write theorems for things we wish to prove, rather than things we have already proven, it would seem that usage of the term "theorem" in PVS more closely follows the second definition given above than the first. It is also possible that this second definition reflects changing English usage, as the definition is more recent than that of the first, though the first is undoubtedly more mathematically rigorous. In general, the theorems we construct in PVS follow a well-known pattern.

```
<Theorem Name> : THEOREM
  <Antecedents>
  IMPLIES <Consequents>
```

By using implication, we can reproduce the functionality of a logical sequent within the context of a Boolean expression. In this manner, `<Theorem Name>` is a unique identifier, `<Antecedents>` is a conjunctive list of all the sequent's premises, and `<Consequents>` is a conjunctive list of all of the logical expressions we wish our sequent to prove. It should be noted that PVS implicitly assumes universal closure, and that any free variables are implicitly universally quantified (Owre et al., 2001). As such, free variables do not need to be universally quantified explicitly, but can be for legibility.

2.2.2 Antecedents

The premises of our sequent must include in an appropriate manner the definitions generated by the translation software, whether directly or indirectly. This includes, but is not limited to, transition predicates, requirement pre-conditions, information regarding relevant previous program states, and overall assumptions. Consider the following theorem, taken from our fourth case study in §6.4.

PVS

```
correctness_1 : theorem
  forall(x1 : ByteEn, x2 : ByteCount, x3 : bool) :
    x3 = True
    and valid_bytemask(x1)
    and valid_bytecount(x2)
      and transition(1, s(0),s(1), x1, x2, x3)
    implies req_word_pointer(x1, x2)
        = outputs_WdPointer_(1,s(1),s(1),x1,x2,x3)
```

END PVS

In the above example, we have examples of general conditions (line 3), miscellaneous predicates (lines 4 and 5), a transition predicate (line 6), requirements predicate (line 7), and output method derived function invocation (also line 7). As we can see, the underlying formulation still holds. A conjunctive list of premises is used to imply some particular conclusion.

Transition predicates, as produced by the BAPIP translation software, must be included as an antecedent for any theorem in order to prove properties of the modules they model, as demonstrated above. The transition predicates BAPIP provides contain all user specified schedules, selectable via the first argument of the predicate. Information concerning these schedules is automatically provided in the top-level generated PVS file. Due to Bluespec's atomic transaction architecture, a Bluespec module's "input wires" are not always available, in a hardware sense, but must be invoked by a parent-level entity. When constructing theorems in PVS, the user is that parent-level entity. When the user invokes the transition predicate, care must be taken, or one risks invalidating the theorem. For example, if the various transition predicates and tabular specifications used are not all given the same input variables, or if the wrong set of methods is selected to execute in a given clock cycle, inconsistent and incorrect results are highly probable. Input values of the specification must be passed to the first transition for them to be modelled, and a schedule must be selected that invokes the input methods appropriate to the specified data.

Though more complex ways to organize the interaction of requirements with the model are described below, the most basic form is to manually extract pre and post conditions from the specification and add them to the theorem. This is the method used in §6.5. In the case of pre conditions, the Boolean expression describing the precondition is added to the conjunctive list of antecedents without any modification. Due to the implicit universal closure of theorems, the pre-condition will act as a restriction on the input space. This process may be made more efficient by taking more advantage of PVS's type system. Such improvements are left as future work.

Depending on the nature of the behaviour under examination, it may be necessary to describe the behaviour of pre-states previous to the pre state used by the transition predicate. One example is if the calculation outcome is dependent not just on the temporally local input, but on values stored in memory. Such preconditions may be encoded as Boolean conditions invoking those variables directly, as state is encoded in nested record syntax, and record field access is an easy enough operation. An alternative that could result in greater efficacy would be to use predicate sub-types. Adaptation of the existing algorithm to use predicate sub-types is left as future work. If the calculation under examination requires more than one clock cycle to calculate the expected value, it is then necessary to "chain together" transition predicates. In this case, one additional transition predicate should be added for every additional previous state that the calculation under examination requires. Determination of the number of previous states required often implies a thorough understanding of the design under verification. Each additional transition should take as pre-state the post-state of the preceding transition, thus describing continuous operation.

It may also sometimes be necessary to make global assumptions about the nature of the inputs. This can result from the discovery of implicit assumptions in informal specifications, discovered during the formalization process, such as those discovered by Pang et al. (Pang et al., 2015). One such example, occurring in §6.1, is the fact that one of the inputs to the function block is required to be greater than zero, though this does not appear in the original documentation (IEC, 2013).

While it is possible to take such pre-conditions as axioms, the vastly preferred method is to include them as additional antecedents, in the same manner as functional pre-conditions. PVS's existing set of axioms is consistent, and adding additional axioms, why syntactically possible, runs the risk of introducing contradiction (Rushby et al., 1998). Additional axioms would need to be demonstrated consistent with the existing PVS axioms, which is not preferable to simply using definitions and the type system. Encoding our hardware designs using definitions and the type system constitutes a conservative extension of PVS's deductive system, so the system as a whole maintains consistency.

2.2.3 Consequents

The process of theorem proving attempts to deduce the consequents of a theorem from its antecedents. As with pre-conditions, we will examine more useful organizations in §5.2, but the basic form of these consequents is the post-condition of the functional requirements. In the case of tabular specifications, the consequent will most often take the form of a test of equality between the variable under examination and the expected value of that variable. If more than one post-condition must hold simultaneously, simply add it to the conjunctive list of consequents. This is the same form given in the example above in §2.2.2.

2.2.4 Using the PVS Interactive Proof Environment

Once a theorem has been constructed, it remains to be demonstrated that it is correct using the formal mathematics of the PVS proof environment. Manual proofs of complex sequents in PVS can be cumbersome, but the theorem constructions presented above are highly amenable to the automatic proof strategies native to PVS.

In order to enter the proof environment from within a loaded PVS instance in Emacs, place the cursor somewhere inside the theorem you wish to prove and invoke the prover with `M-x prove` or `C-c p`. PVS will then typecheck the theory and any imported theories, and upon successful completion open a new buffer in split-screen mode. This new buffer will contain the PVS proof environment, which may be interacted with by entering commands and strategies at the prompt. The goal of this process is the discharging of all proof obligations generated from the theorem under examination. Upon successful completion, the prover will display `Q.E.D.`, and transcribe the proof generated into an associated `.prf` proof file.

2.2.5 Automatic Deduction on Functional Requirement Sequents

Both theorems for which pre and post conditions have been manually entered and theorems which use tabular specifications are proven in the same manner. In simple cases, an invocation of the `(grind)` command will discharge the proof, with no intervention required from the user. In some cases, rewriting a large number of unnecessary recursive definitions may bog down the prover, causing unacceptably long run times. In such cases, it is recommended the user instruct the strategy not to automatically install recursive definitions as auto-rewrite rules. In order to do this, use of the modified command `(grind :defs explicit)` is recommended.

It is sometimes the case that `(grind)` is not adequate in and of itself to discharge a proof with a high level of complexity, such as the theorem presented in §6.5. This might be due to either the size of the underlying modules, or the requirement for multiple chained transitions. In these cases, a certain amount of proof decomposition in the initial phases is necessary before proceeding with the Swiss-Army sledgehammer that is `(grind)`. A concrete example is presented in §6.5, but the general strategy is as follows. Composition of a new automatic PVS strategy based on this general outline may compose interesting future work.

1. `(skolem!)` - Skolemization to eliminate the universal quantification (whether explicit or due to universal closure).

2. `(bddsimp)` - A rudimentary algorithm for binary decision diagram simplification. This breaks the sequent into any relevant top-level subproofs.

3. `(expand *)` - Takes function and predicate definitions and applies them throughout the sequent.

4. `(lift-if)` - Exposes conditional expressions as top-level sub-sequents.

Steps 3 and 4 are applied interchangeably as needed. Although the point at which a sub-proof becomes grindable is not yet well known, repeating this procedure does typically result in a grindable proof sequent.

2.2.6 Automatic Deduction on Consistency Sequents

Theorems attempting to prove the consistency of our transition predicates require a slightly different tactic to discharge than our functional requirement theorems, owing to the presence of existential quantification. Existential quantification cannot be addressed via automatic proof strategy in PVS. In order to prove existence, PVS requires the proof operator to provide a hypothetical instantiation value, which is substituted into the sequent in place of the quantified variable. In anticipation of this requirement, the BAPIP translator automatically generates modified transition functions which may be used for this instantiation step. This instantiation term consists of the transition predicate, modified to be a function returning the post-state of the transition, rather than being a predicate on the equality of the calculated post-state and the post-state provided as an argument to the predicate. As such, the function has no post-state argument. The following commands, executed in sequence, will successfully discharge consistency theorems, presuming the transition predicates under analysis are in fact consistent.

```
(skolem!)
(inst + < instantiation term >)
(expand transition)
(expand transition_val)
(assert)
```

Where the `< instantiation term >` is an invocation of the modified transition function, using skolemized variables for arguments. These proofs are sufficiently routine that BAPIP automatically generates prooflite scripts containing proofs discharging them, conforming to the methodology specified above. The following is an automatically generated theorem and prooflite script from §6.2.

PVS

```
%|- consistency_0 : PROOF
%|- (then (skolem!)
%|-       (inst + "transition_val (i!1, pre!1)")
%|-       (rewrite transition)
%|-       (rewrite transition_val)
%|-       (assert))
%|- QED
```

```
consistency_0 : Theorem
  FORALL (i : nat, pre : mkAlrm_int) :
    EXISTS (post : mkAlrm_int) :
      transition(i, pre, post)
```

END PVS

As we can see, the instantiation term is simply a version of the transition predicate returning a concrete state, rather that comparing a pre and a post state. This theorem asks the question “for every input state, there must exist a valid output state.” To prove the existence of a valid output state, the easiest thing to do is provide one, which satisfies the requirement for existential quantification.

2.3 A Previous Monadic Embedding

The original basis for the work herein presented was a paper by Richards and Lester (2011), which demonstrated a manual method for the embedding of Bluespec SystemVerilog in the higher order logic of PVS. This system employed numerous library functions, as well as a `BSVMonad` type, in order to minimize the syntactic difference between the Bluespec guarded action structure and the corresponding PVS embedding. The intention was to minimize the difficulty of performing the transformation manually. Recognizing the numerous advantages of an automated process over a manual one, the original intent of the research presented in this thesis was to merely automate the manual embedding. When the time came to apply the newly created program, then called BSV2PVS, to practical examples, however, several deficiencies were recognized in the original work which prevented its application to more complex problems.

The most fundamental issue relates to the timed and untimed semantics discussed in §4.1. The Richards and Lester embedding, as a byproduct of attempting to reproduce the action-oriented structure of BSV, modelled each action as an individual transition predicate. Transition predicates for the entire module are modelled as the disjunction of all action transitions. They modelled a transition relation, rather than a transition function, as this setup can result in multiple contradictory, but still valid, post states. In short, they go only so far as to model the untimed semantic, as the effort towards modelling the timed semantics are incomplete. In only attempting to model the deterministic but arbitrary choice of the Bluespec scheduler, the complexities of the Bluespec scheduler's action arbitration semantics are completely ignored. This severely restricts the applicability of the Richards and Lester method. Since no deterministic decision is made about the post-state, chaining transition predicates together to model behaviour over more than one clock cycle becomes inherently problematic. The number of states this produces is the number of possible states in one clock cycle raised to the power of the number of clock cycles we have to model over, which exceeds the practical limits of computation very quickly. Further, it is not even a good model of the underlying hardware, as the hardware is necessarily deterministic, as it is mapped to a physical system.

It also fails to model a very basic function of Bluespec modules, in that modules will not do anything if there is nothing to be done. If no methods are invoked and no rules are active during a particular clock cycle, a Bluespec module will not change its state or perform calculations, other than those required to determine whether or not its rules are active. Since the Richards and Lester transition relation is the disjunction of all action transitions, taking the transition relation as an antecedent specifies that at least one action must have been executed, though again, it is impossible to know which one. We explore this concept further in §4.2.6 by comparing our method to the Richards and Lester method as applied to a running example.

Furthermore, by not modelling the complete timed semantic, a Richards and Lester transition relation at best models an untimed step, as will be discussed in §4.1 Untimed steps have no direct relationship with hardware clock cycles, so it is impossible for such a transition relation to make any assertions at all about timing requirements.

By contrast, our method, encoded as the BAPIP translation tool, accurately models the timed semantic of BSV hardware descriptions. Our transition functions deterministically calculate a module's output from its state and inputs. Multiple transition functions may be chained together to model behaviour over multiple clock cycles. We impose no artificial restrictions on whether at least one action must occur in any given clock cycle. Further, our transition predicates map to actual hardware clock cycles, allowing us to test claims of timing properties in hardware descriptions.

2.4 RISC V and RapidIO

The primary case studies provided in this work (§6.3 & 6.5) are applied to a Bluespec SystemVerilog implementation of the RapidIO message packet passing subsystem of the RISC-V processor Instruction Set Architecture (ISA) available in the Shakti processors project, by (Gala et al., 2016). It is therefore appropriate to introduce the RISC-V processor, the Shakti project, and provide an overview of the RapidIO framework.

2.4.1 RISC-V

RISC-V is an open-source specification for the design of computer processors (Porter III, 2018). Reduced Instruction Set Computer (RISC) stands for Reduced Instruction Set Computer, and RISC-V is an open-source hardware initiative led by the RISC-V Foundation (2020). In order to understand the purpose of RISC-V, one must consider the importance of ISAs in the design of computers. Processing, as performed in Von Neumann architectures, comprises the manipulation of memory through the execution of instructions. The set of instructions a processor implements, and the manner of their implementation, are the objects of a great deal of design effort on the part of processor manufacturers. Each instruction that is included must have corresponding electronic circuitry within the final processor. The set of instructions that a processor implements is that processor's ISA, and there are multiple ISAs used by different manufacturers, for example, Intel's ubiquitous x86-64, or Zilog's Z80, which was highly popular in embedded systems applications in the 1980s, such as audio synthesizers and arcade machines. CISCs, or Complex Instruction Set Computers, often have large numbers of instructions. This is sometimes due to the need for backwards compatibility with legacy software. One of the primary design goals of the RISC-V architecture (and all other Reduced Instruction Set Computer architectures, of which there are many) is to implement a minimized set of instructions, decreasing the size, cost, and design effort required to implement the processor. (Porter III, 2018) There is a trade-off between the number of instructions in an Instruction Set Architecture (ISA) and the amount of memory the controller requires. Reducing the number of instructions a Central Processing Unit (CPU) can execute necessary reduces the expressivity of the associated machine language. Given some complex operation, a RISC must use at least as many instructions as a CISC, and possibly many more, because limiting the number of instructions limits the expressivity of the assembly language. Not only do more instructions take more CPU cycles to execute, but the program itself will tend to be larger, requiring more instruction memory. In the early days of computing, when memory was expensive, the economic incentive was to build up the CPU so that less memory would be required. As memory has become less expensive and CPU speed has increased over the decades, the RISC approach is generally favoured, and now Intel and AMD are supporting Complex Instruction Set Computer (CISC) ISAs with RISC-style micro-instructions (Isen et al., 2009).

Another point of difference between RISC-V and other ISAs is that it is an open-source specification. For many processor manufacturers, precise details about the implementation of specific instructions are a closely guarded secret, protected by intellectual property law, licensing fees and non-disclosure agreements. If this were not the case, any company could manufacture chips based on the x86-64 ISA, for example, and create undesirable competition for Intel. However, many companies and individuals are now recognizing open-source licensing as a viable alternative to this standard industrial practice. Under open-source these designs are much easier to use and contribute to, meaning the development of the design can leverage a far greater pool of talent than any one company could hope to, irrespective of their size and respectability. As such, RISC-V makes itself much more attractive to researchers and industrialists, who can engage with the ISA without incurring prohibitive licensing costs, producing implementations and improvements. This creates an “ecosystem” of RISC-V processors, rather than a handful of strictly regulated sources.

2.4.2 The Shakti Project

The Shakti project is a family of implementations of the RISC-V ISA by (George et al., 2018), and the RISE group at IIT Madras. The work is currently ongoing, and individual cores within the family have different design focuses, such as targeting embedded, control, and mobile processor applications, and security-focused and fault tolerant variants. The Shakti family of processors have been designed and implemented in Bluespec SystemVerilog, citing a higher level of abstraction, “superior behavioural semantics,” architectural transparency and parameterizability as justification. Bluespec Inc. has itself recently pivoted towards being a supplier of RISC-V technologies (Bluespec Inc., 2019). Additionally, Bluespec has seen an open-source release on github (`https://github.com/B-Lang-org/bsc`), with it's first open source release in July of 2021.

Because Shakti is an open source project, the source code for these processors and their subcomponents is fully available for viewing and download (Madhusudan, 2018). As such, the BSV designs were available for use as verification case studies. The combination of open source code and the free availability of the standards from which it was derived, as well as the non-trivial size of the example, made Shakti ideal for a demonstration of our techniques.

2.4.3 The RapidIO Interconnect Framework

The particular subsystem focused on in our case studies is the RapidIO message passing system. Generally speaking, the various chips of a circuit board require some sort of framework for the transmission of information. There are many approaches to this problem; the one used by Shakti is RapidIO, an open standard.

The RapidIO system passes information through its network via message packets. These packets often represent instructions to be carried out or requests for data. The format of the message packets is rigidly determined by the specification (RapidIO.org, 2017), and so is the format of response packets.

The RapidIO subsystem under examination is that which forms response packets in response to packets received from the network. The focus of the verification case studies presented in this thesis, at a high level, is to verify that the response packets formulated by the Shakti BSV implementation of RapidIO are consistent with various properties derived from examination of the RapidIO specification.

2.5 Extended Backus Naur Form

Throughout the work to follow, EBNF (Jensen and Wirth, 2012) will be used to formally and systematically encode syntax, be it BSV, PVS, or any other language. This system will be used mainly to define valid parsing targets for the BAPIP tool.

EBNF presents sets of syntax rules, which may be used symbolically and recursively in further definitions. A *rule of syntax* is an arrangement of strings and syntax rules which define correct grammatical arrangements of

Symbol	Description
$\langle \ldots \rangle$	Syntax Rule (Non-terminal) Symbol
$\langle \ldots \rangle$ ::=	Syntax Rule Definition
$\vert$	Alternative
'...'	String Delimiters (Terminal Symbol)
[...]	Option Delimiters
{...}	Repetition Delimiters (Zero or more iterations)

Tab. 2.1: EBNF reference table

lexemes in the target language. Rules of syntax may have multiple valid definitions, indicated by |. Strings are the atomic element of EBNF, and will be delimited by single quotes. Some constructions may be optional, and are delimited by square braces. Other constructions may occur a number of times equal to or greater than zero, and are delimited by curly braces. Concatenation is implicit. For a summary of EBNF symbols, please see Table 2.1.

2.6 Primer on Haskell and the Haskell Type System

Being the language of implementation for the translation software BAPIP, an introduction to Haskell is necessary for understanding various code snippets within this thesis.

2.6.1 Haskell Type System Primer

It is helpful at this point to review the notation used in the construction of Haskell types. The definition below will be provided using EBNF, as described in §2.5.

GRAMMAR

⟨*Type Definition*⟩ ::= '`type`' ⟨*Type*⟩ '`=`' ⟨*Type*⟩
  | '`data`' ⟨*Type*⟩ '`=`' ⟨*Constructor List*⟩ ['`deriving(`' ⟨*Typeclass List*⟩ '`)`']
  | '`data`' ⟨*Type*⟩ '`=`' ⟨*Constructor Name*⟩ '`{`' {⟨*Field List*⟩} '`}`' ['`deriving(`' ⟨*Typeclass List*⟩ '`)`']

$\langle$*Constructor List*$\rangle$ ::= $\langle$*Constructor*$\rangle$ { '`|`' $\langle$*Constructor*$\rangle$ }

$\langle$*Constructor*$\rangle$ ::= $\langle$*Constructor Name*$\rangle$ { $\langle$*Type*$\rangle$ }

$\langle$*Field List*$\rangle$ ::= $\langle$*Identifier*$\rangle$ '`::`' $\langle$*Type*$\rangle$ {, $\langle$*Identifier*$\rangle$ '`::`' $\langle$*Type*$\rangle$ }

$\langle$*TypeClass List*$\rangle$ ::= $\langle$*TypeClass*$\rangle$ {'`,`' $\langle$*TypeClass*$\rangle$}

END GRAMMAR

The first type of type definition is a "type synonym." This is a simple renaming of a type, and is generally used for code readability and ease of coding. These can sometimes be fairly complex types including tuples and lists in any combination.

Haskell's type system also has an "or" symbol, denoted by the '|' character, which functions very similarly to the same operator in EBNF. Specifically, it denotes that a type can be constructed with any one of the specified type constructors. These type constructors can be pattern matched against, which can be used to determine which type constructor a value of this type has used at any given time. The third definition is record syntax. This functions somewhat similarly to C's `struct`, except that, in typical Haskell style, implementation details are handled for us, and the "labels" for the fields of the record are actually also names of functions which take the record as an argument and return the corresponding field as a result. The populated type structure is the generated result of the parser, and therefore hypothetically minimizes the number of semantic details included. This structure tries to be an encoding, not an interpretation.

2.6.2 The Anatomy of a Haskell Function

Haskell is a purely functional language. Functions are first class members, and function application is so common in Haskell, arguments are applied to functions through mere juxtaposition, without requiring parenthesis as in most languages. Effectively, the space character applies an argument to a function. The purpose of this section is to provide a general overview of how to read a Haskell function.

```
<function name> :: <Type Signature>
<function name> <pattern A> = <Expression A>
<function name> <pattern B> = <Expression B>
<function name> <pattern C> = <Expression C>
```

The first line of a Haskell function declares the type signature of the function. Type signatures are discussed below. During evaluation, a Haskell function will attempt to pattern match on each of the provided patterns. The expression corresponding to the first matched pattern will be evaluated, and the result will be the return result of the function. Pattern matching is also used to label inputs with identifiers, which may change from pattern to pattern. One common form in a Haskell function is to split a list into its first element and all subsequent elements using pattern matching, perform some operation on the first element, and then recursively call the function on the remaining elements. Note that this is different from tail recursion. For example, the following Haskell function computes the sum of a list.

HASKELL

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

END HASKELL

The type signature above indicates the function expects a list of integers, and produces an integer. Haskell functions may accept any number of arguments, but those arguments are strongly statically typed. In the author's experience, when developing a Haskell program, more care, attention, and work is required to ensure your program is correctly and consistently typed, but as a result, Haskell programs require far less debugging of semantic errors.

The base case for this recursion is the empty list `[]`. The recursive case uses the list construction "cons" operator `:` to distinguish the head of the list (`x`) from the tail (`xs`). The returned result is x, added to the result of `sum` called on the tail of the list.

A slightly more complex example is the quicksort algorithm implemented in Haskell. The fact quicksort can be written in so few lines in Haskell, and so lucidly, is a testament to Haskell's power as a language.

HASKELL

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = lowerList ++ x ++ higherList
  where
    lowerList = quicksort [l | l <- xs, l <= x]
    higherlist  = quicksort [h | h <- xs, h > x]
```

END HASKELL

In Haskell, `++` is the concatenation operator, and the parts occurring between square braces are list comprehensions. One modification which is often made for readability is the assignment of temporary values using `where` clauses. Lines following the `where` clause are local bindings enhancing legibility.

Most of the functions composing the translation software feature such clauses. Concepts such as control flow and sequential execution of operations do not figure prominently in the Haskell programs in this thesis.

2.7 Tabular Specifications

Tabular Specifications, in particular function tables, are a formal means of describing the behaviour of some output variable in a system, based on inputs to said system. One tabular specification, presented in §6.1, is as follows:

Condition	**QH**
$X > H$	True
$(H - EPS) \leq X \leq H$	No Change
$X < (H - EPS)$	False

The interpretation of the specific symbols in the above specification is best left for §6.1. If we generalize the above, we get the following form.

Condition	Ψ
$P_0(\Phi)$	ψ_0
$P_1(\Phi)$	ψ_1
$P_2(\Phi)$	ψ_2
$\vdots$	$\vdots$
$P_3(\Phi)$	ψ_n

Where:

- Φ is the set of input variables.
- Ψ is the output variable.

- P_i is a predicate on the input variables.
- ψ_i is some element of the domain of Ψ.

The way that the table is read, is that, if we consider some set of particular inputs to the function or module represented by the table, we evaluate which of the available predicates is true. The ψ_i on that row is then assigned as the value of the output variable Ψ. A question that naturally arises from this description would be what happens when multiple predicates are true simultaneously. We will see that with a correctly formulated table, such an event is impossible, due to the property of disjointness. A table is not considered properly constructed, unless it is both complete and disjoint.

By encoding a tabular specification in PVS using the `TABLE` construct, the PVS theorem prover may be used to automatically prove the completeness and disjointness of the specification.

2.7.1 Disjointness

A table is considered *disjoint* iff, for every pair of predicates, it is impossible for both to evaluate to true simultaneously. Or:

$$\neg\exists\phi \bullet \ (\forall p, q \in P \bullet \ p \neq q \wedge \neg(p(\phi) \wedge q(\phi)))$$

That is, there does not exist a set of inputs for which both of some distinct pair of predicates in the set of predicates evaluate true. For example, for $x \in \mathcal{N}$, the predicates $p(x) = x < 6$ and $q(x) = x > 8$ are disjoint, because x can not simultaneously be greater than 8 and less than 6. The expressions $p(x) = x < 22$ and $q(x) = x > 14$ are not disjoint, because there exist values of x which satisfy p and q simultaneously.

2.7.2 Completeness

The completeness property might also be termed a totality property, in that a tabular specification that is not complete is a partial function, and a tabular specification which is complete is a total function from Φ to Ψ

The completeness property is as follows:

$$\forall\phi \in \Phi \bullet \ P_0(\phi) \vee P_1(\phi) \vee P_2(\phi) \vee \cdots \vee P_n(\phi)$$

In other words, for all particular sets of input values, one of the predicates P_i must hold.

2.7.3 Complications

In addition to the above descriptions, there are ways tabular specifications are often abbreviated, which bear mentioning, since they require some degree of semantic interpretation. Consider the following tabular specification:

Condition	Ψ
$X \wedge Y$	ψ_0
$X \wedge \neg Y$	ψ_1
$\neg X \wedge Y$	ψ_2
$\neg X \wedge \neg Y \wedge Z$	ψ_3
$\neg X \wedge \neg Y \wedge \neg Z$	ψ_4

We may notice, from the above, that all our predicates are the conjunction of the two terms X and Y, or their negations. We may visually separate the table along this conjunction:

Condition			Ψ
X	Y		ψ_0
X	$\neg Y$		ψ_1
$\neg X$	Y		ψ_2
$\neg X$	$\neg Y$	Z	ψ_3
$\neg X$	$\neg Y$	$\neg Z$	ψ_4

It is a usual simplification to eliminate blank cells in the table by merging them with the cell to the left, as so:

Condition			Ψ
X	Y		ψ_0
X	$\neg Y$		ψ_1
$\neg X$	Y		ψ_2
$\neg X$	$\neg Y$	Z	ψ_3
$\neg X$	$\neg Y$	$\neg Z$	ψ_4

Our table still contains some unnecessary repetition. We can collect adjacent identical conditions vertically as follows:

Condition			Ψ
X	Y		ψ_0
	$\neg Y$		ψ_1
$\neg X$	Y		ψ_2
	$\neg Y$	Z	ψ_3
		$\neg Z$	ψ_4

The above simplifications are designed to increase the readability and ease of use of tabular specifications. This is accomplished through the removal of redundant formulas.

3. DEFINITION AND PARSING OF OUR SUBSET OF BSV

In this chapter, we will examine the translation algorithm. Complete syntax both for accepted BSV hardware descriptions and generated PVS files will be presented. A semantic interpretation of our abstract syntaxes, and the mapping between the two, will also be presented.

The syntax presented in this section represents only those language constructs selected for inclusion in the translation. Aside from some basic elements, such as literals, types and arithmetic expressions, language constructs qualified for inclusion based on their presence in targeted case study descriptions. This includes enough language elements to be practical for design purposes, while avoiding some of the more exotic features that would be required by a full translation, such as the ability to include C functions in BSV modules (Bluespec Inc., 2012a).

3.1 *Defining a Grammar for BSV*

In order to parse a BSV file, we must first define a grammar. This will be accomplished in bottom-up form using Extended Backus-Naur Form (EBNF). For more information, please see §2.5. The Parsec parsing library (Leijen and Meijer, 2001) resembles EBNF syntactically, so the grammar presented in this section resembles the actual source code of the parser. Please note that this is a syntactic definition only. The semantic interpretation of these syntactic constructs is discussed in §3.3. For a thorough discussion of the semantic translation, please see §4.

It must be stressed that this grammar defines a subset of BSV which is amenable to translation via BAPIP. BSV has a large number of language constructs which the translation software makes no attempt to translate, so we will provide here a positive description of the supported sub-language. Historically, this subset was originally developed as a small subset of the language accepting the decriptions in case studies 1 and 2 (§6.1 and §6.2).

Language features were added gradually as demanded by the ongoing work towards developing case studies.

The grammar presented here has been partially derived from the Bluespec reference manual (Bluespec Inc., 2012a), and partially re-engineered over the course of the project. At the time when this portion of the work was taking place, the Bluespec compiler was a closed-source tool, thus requiring some degree of reverse engineering. The compiler has since been released open source (Bluespec Inc., 2020).

Throughout this section, a modified version of the BSV program from §2.1 will be used to demonstrate the applicability of the grammar being discussed. This will be demonstrated in the "running example" section of all relevant subsections. This BSV description has been expanded, so that each of the specified grammar productions is represented within the description. The circuit which can be derived from this example is not meaningful or useful. This example is meant purely as a demonstration of syntax.

Top-Down Overview of Bluespec Grammar

Although BSV files may contain several packages, the package is the largest working unit of the Bluespec language. Packages primarily contain module declarations, import statements, and various declarations (type definitions, constants, functions, etc.). Modules are the top-level construct which can be wholly synthesized into hardware. Modules are broken down primarily into state declarations and various flavours of action. Actions are the only elements which carry statements.

For an pictorial overview of structure of BSV grammar, see Figure 3.1. Identifiers, type annotations and literals have been omitted from this figure. Including these would not add to our understanding of the structure of the language, and they clutter the diagram quite badly.

To reduce clutter, syntactic units which were very common have been omitted (specifically identifiers, type annotations, and literals).

Haskell Data Structures

After parsing, the generated abstract syntax tree is encoded in Haskell data types. Since Haskell is statically typed, the type structure cannot be violated, providing assurance that the data extracted from our BSV descriptions aren't misused. In the grammatical descriptions which follow, the Haskell data

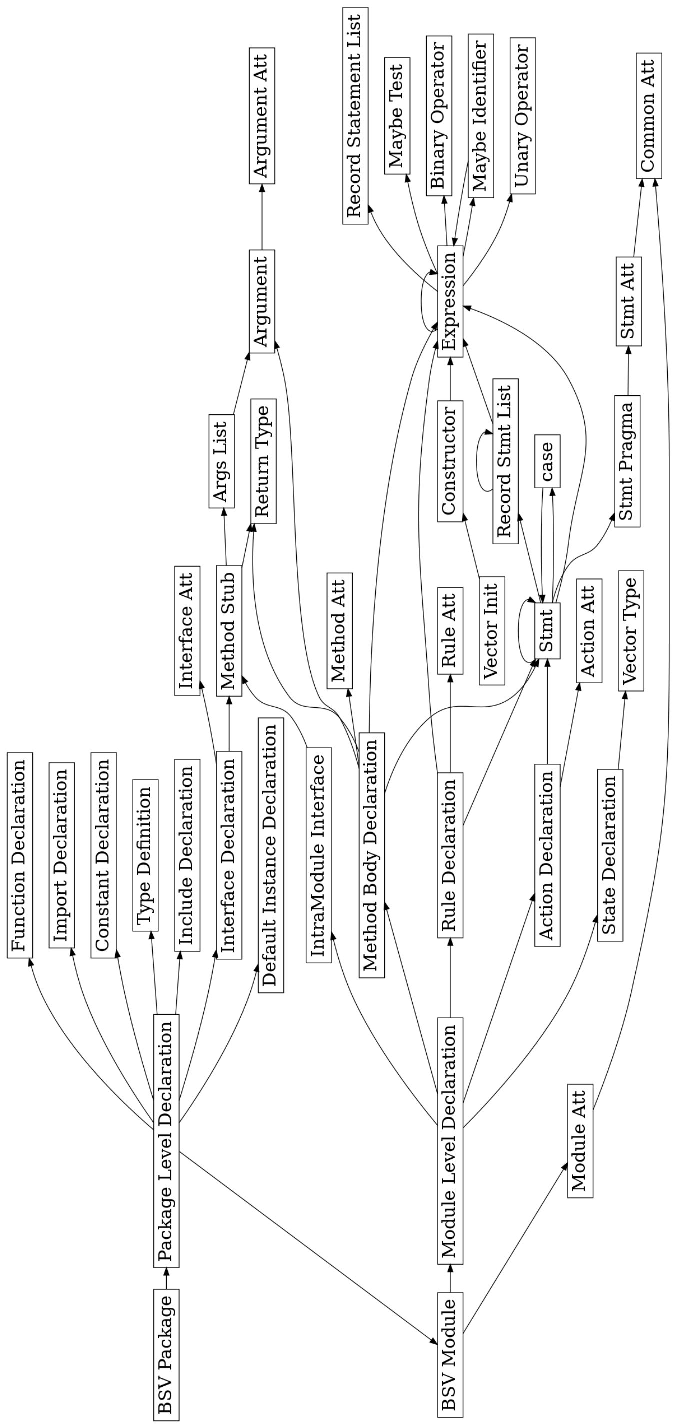


Figure 3.1: BSV Grammar Hierarchy Diagram

types which encode the given grammar will be stated, with explanatory notes. For a primer on the Haskell type system, see §2.6.1.

Comments and Whitespace Handling

Comments in BSV follow the C convention, with line comments indicated by `//`, and block comments enclosed by `/*` and `*/`. All comments are stripped out during preprocessing. Neither comments nor whitespace are included in the following syntactic description, and no arrangement of whitespace and comments will prevent a description from being parsed.

Populating the PVS design with BSV comments would be a useful addition to the process, hypothetically increasing the readability of the output. However, in practice this would only be applicable to state elements and methods. Rules, modules, interfaces, and packages are either flattened by the translation process or have no analog in the PVS output, so comments would not be locatable within the output files.

Macro Definitions and the BAPIP Preprocessor

In addition to stripping whitespace, BAPIP preprocesses macros out of a BSV file prior to invoking the parser. Although BSV supports macros in the C style, macros as defined in PVS operate very differently. A PVS macro behaves like a constant declaration, whereas a BSV macro may contain arbitrary characters, including expressions and commands.

Since there is no direct translation of macros possible, macros are made the subject of preprocessing. All macros (which typically occur in included `.define` files), are collected, and a find-and-replace operation is executed over each BSV file to be processed. At the time of this software's design and implementation, the Bluespec compiler was closed source, so such routines were inaccessible. At any rate, the macro substitutions themselves were trivial to implement.

3.1.1 BSV Types

Due to its nature as a hardware description language, numeric types which do not have an implicit bitwidth (such as `Float` and `Bool`) are parameterized by bitwidth. In other HDLs, this information is attached to registers and wires, but in BSV it is incorporated into the type system, and adherence to bitwidth limitations is enforced by the Bluespec type-checker.

GRAMMAR

⟨*BSV Type*⟩ ::= '`Bool`' | '`Float`' | ⟨*Identifier*⟩
| '`Bit#(`' ⟨*decimal digit*⟩ {⟨*decimal digit*⟩} '`)`'
| '`Int#(`' ⟨*decimal digit*⟩ {⟨*decimal digit*⟩} '`)`'
| '`UInt#(`' ⟨*decimal digit*⟩ {⟨*decimal digit*⟩} '`)`'
| '`Maybe#(`' ⟨*BSV Type*⟩ '`)`'

END GRAMMAR

Abstract Syntax in Haskell

The parameterizability of data types in BSV transfers effectively into Haskell's type system. A BSV type in Haskell may be any of the types given above. The custom data type is for type definitions and enumerations, and corresponds to the ability of an identifier to form a type name in BSV. Numeric types are parameterized by a bit width, and the Maybe type is parameterized by another BSV type. In theory this makes the type system divergent, but since all BSV modules are assumed to have appeased the BSV compiler before having reached BAPIP, this is not an issue in practice.

HASKELL

```haskell
data BSVType = BSV_Bool -- Booleans
  | BSV_Bit N             -- Bit Vector
  | BSV_Int N             -- Signed Integer
  | BSV_UInt N            -- Unsigned Integer
  | BSV_Real              -- Floating Point
  | BSV_Custom Name       -- As defined in typedef
  | BSV_Maybe BSVType -- Maybe monad
deriving (Eq, Show, Ord)

type N = Integer
```

END HASKELL

The Maybe Type

The `Maybe` type in BSV is a type which encapsulates other types, and seems to be a holdover from when BSV was a library of Haskell. Things of type `Maybe` may be either valid or invalid. This terminology differs from normal usage for Maybe types in other languages, but is taken from the BSV documentation (Bluespec Inc., 2012a). If a value is valid, then some specific concrete value is coupled to it. Otherwise no value is coupled. This ostensibly allows us to save the trouble specifying a value for the invalid case, and allows a degree of pattern matching from within if statements.

3.1.2 Literals

`Literals`, given below, are much as one would expect them in a C-based language. The underscore character is a valid syntactic character, but has no semantic meaning. It is used for readability and formatting purposes to assist in keeping place value straight. For example, `16b1010101010101010` can be written as `16b1010_1010_1010_1010`. In BSV the ability to designate size literally and numerically, rather than implicitly via data types (long or short int, etc.) is essential to BSV's function as a hardware description language.

GRAMMAR

⟨*Literal*⟩ ::= ⟨*Real Literal*⟩ | ⟨*Integer Literal*⟩ | ⟨*String Literal*⟩
 | ⟨*Boolean Literal*⟩ | ⟨*Enumeration Literal*⟩ | ⟨*Other Literal*⟩

⟨*Integer Literal*⟩ ::= ‘`'0`’ | ‘`'1`’
 | ⟨*decimal digit*⟩ ⟨*decimal digit*⟩ ⟨*based Literal*⟩
 | ⟨*decimal digit*⟩ ⟨*decimal digit*⟩

⟨*based Literal*⟩ ::= (‘`d`’ | ‘`D`’) ⟨*decimal digit*⟩ {⟨*decimal digit*⟩}
 | (‘`h`’ | ‘`H`’) ⟨*hexadecimal digit*⟩ {⟨*hexadecimal digit*⟩}
 | (‘`o`’ | ‘`O`’) ⟨*octal digit*⟩ {⟨*octal digit*⟩}
 | (‘`b`’ | ‘`B`’) ⟨*binary digit*⟩ {⟨*binary digit*⟩}

⟨*Real Literal*⟩ ::= ⟨*decimal digit*⟩ { ⟨*decimal digit*⟩ } ‘`.`’ ⟨*decimal digit*⟩ { ⟨*decimal digit*⟩ } [(‘`e`’ | ‘`E`’) [‘`-`’] ⟨*unsized Integer Literal*⟩]

⟨*String Literal*⟩ ::= ‘`"`’ { *character* } ‘`"`’

⟨*Boolean Literal*⟩ ::= ‘`true`’ | ‘`True`’ | ‘`TRUE`’ | ‘`false`’ | ‘`False`’ | ‘`FALSE`’

⟨*Enumeration Literal*⟩ ::= ⟨*Identifier*⟩

⟨*decimal digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’ | ‘`_`’

⟨*hexadecimal digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`8`’ | ‘`9`’ | ‘`a`’ | ‘`b`’ | ‘`c`’ | ‘`d`’ | ‘`e`’ | ‘`f`’ | ‘`A`’ | ‘`B`’ | ‘`C`’ | ‘`D`’ | ‘`E`’ | ‘`F`’ | ‘`_`’

⟨*binary digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`_`’

⟨*octal digit*⟩ ::= ‘`0`’ | ‘`1`’ | ‘`2`’ | ‘`3`’ | ‘`4`’ | ‘`5`’ | ‘`6`’ | ‘`7`’ | ‘`_`’

⟨*Other Literal*⟩ ::= ‘`defaultValue`’ | ‘`default`’

END GRAMMAR

Abstract Syntax in Haskell

In Haskell, literals are encapsulated by the `Lit` type.

HASKELL

```haskell
data Lit = LitString String -- Strings
  | LitEnum String          -- Enumerats
  | LitInt  Integer         -- Integers
  | LitBool Bool            -- Booleans
  | LitReal Float           -- Floating Point
  | LitChar Char            -- Characters
  | LitSizedInt  N Integer  -- Integer with bit width
  | LitStructConstructor    -- Struct  Initializer
  | LitVoid                 -- Internal use
deriving (Eq,Ord)
```

END HASKELL

Structures which are initialized to their default values behave syntactically as literals.

3.1.3 Identifiers

Depending on the context, `Identifiers` must or must not be capitalized. Following the Haskell convention, capitalized identifiers typically indicate types, and lowercase identifiers may be state elements, rule names, etc. Within the context of this grammar, identifiers which must be capitalized are called "Name," and those which must not be capitalized are called "Identifier."

Identifiers and names with leading underscores, while valid syntax in BSV are not in PVS. PVS does, however, allow identifiers to start with Unicode characters other than the ASCII symbol set (Bluespec Inc., 2012a; Owre et al., 2001). Thus, during the translation process, any identifiers or names with leading underscores have those swapped with double low line, a character resembling, but distinct from, underscore. Since BSV does not allow Unicode characters, there is no intersection between the set of allowable BSV identifiers and the set of PVS identifiers thus modified.

GRAMMAR

$\langle Name \rangle$ ::= $\langle UC \rangle$ {$\langle character \rangle$}

$\langle Identifier \rangle$ ::= $\langle LC \rangle$ {$\langle character \rangle$}

$\langle character \rangle$ ::= $\langle UC \rangle$ | $\langle LC \rangle$ | $\langle DD \rangle$ | '_' | '$'

$\langle UC \rangle$ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |

$\langle LC \rangle$ ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

$\langle DD \rangle$ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

END GRAMMAR

Abstract Syntax in Haskell

The Haskell representation of certain low-level constructs, such as identifiers and expressions, are shared between the PVS and BSV abstract syntax trees. One of the necessary translation processes is annotating identifiers with absolute addressing within the PVS state record that will eventually be generated. This, and other instances, such as structure field access, cause the discrepancy between the grammar of identifiers in BSV and the generated data structure in Haskell.

HASKELL

```haskell
data ID_Path = ID_Submod_Struct ModuleInst ID_Path
  | ID String
  | ID_Vect String Index deriving (Eq, Ord)

type ModuleInst = String
type Index = Expression
```

END HASKELL

3.1.4 Expressions

`Expressions` in BSV, for which the grammar is given below, are constructed similarly to C-based languages, and fortunately, are very similar to PVS expressions. Some operations have different symbols, but are semantically identical. Other operators are introduced by the fact that this is a hardware description language, such as those involving square braces. In hardware design, treating variables as bit-vectors and performing sub-variable individual bit access and ranged bit access is quite common, as is bit vector concatenation. These three bit vector operations are included in BSV as primitives.

GRAMMAR

⟨*Expression*⟩ ::= ⟨*Expression*⟩ ⟨*Binary Operator*⟩ ⟨*Expression*⟩
  | ⟨*Unary Operator*⟩ ⟨*Expression*⟩
  | '`(`' ⟨*Expression*⟩ '`)`'
  | ⟨*Identifier*⟩ '`.`' ⟨*Identifier*⟩
    '`(`' [⟨*Expression*⟩ { '`, `' ⟨*Expression*⟩ }] '`)`'
  | ⟨*Identifier*⟩ '`(`' [⟨*Expression*⟩ { '`, `' ⟨*Expression*⟩ }] '`)`'
  | ⟨*Identifier*⟩
  | ⟨*Literal*⟩
  | '`(`' ⟨*Expression*⟩ ') ?' ⟨*Expression*⟩ '`:`' ⟨*Expression*⟩
  | ⟨*Identifier*⟩ '`[`' ⟨*Expression*⟩ '`:`' ⟨*Expression*⟩ '`]`'
  | ⟨*Identifier*⟩ '`[`' ⟨*Expression*⟩ '`]`'
  | '`matches tagged`' ⟨*Maybe Test*⟩
  | '`{`' ⟨*Expression*⟩ { '`,`' ⟨*Expression*⟩ } '`}`'
  | '`tagged`' ⟨*Maybe Identifier*⟩
  | ⟨*BSV Type*⟩ '`{`' ⟨*Record Statement List*⟩ '`}`'
  | '`fromMaybe (`' ⟨*Expression*⟩ '`,`' ⟨*Identifier*⟩ '`)`'

⟨*Binary Operator*⟩ ::= '`==`' | '`!=`' | '`>=`' | '`<=`' | '`>`' | '`<`' | '`&&`' | '`||`' | '`&`' |
    '`|`' | '`^`' | '`<<`' | '`>>`' | '`*`' | '`/`' | '`%`' | '`+`' | '`-`'

⟨*Unary Operator*⟩ ::= '`-`' | '`+`' | '`!`'

⟨*Maybe Test*⟩ ::= '`Valid`' '`.`' ⟨*Identifier*⟩
  | '`Invalid`'

⟨*Maybe Identifier*⟩ ::= '`Valid`' ⟨*Expression*⟩
  | '`Invalid`'

END GRAMMAR

Abstract Syntax in Haskell

As with most expression systems, we must account for a large number of terms, making this one of the largest declarations in terms of lines of code. Again, the expression system may theoretically contain infinitely regressive terms, but unlike the type system, this expression system does not diverge.

HASKELL

```
-- | Expressions are collections of tokens indicating mathematical
-- | operations. These include Boolean operations such as equality
-- | and other comparisons, Bitwise operations, arithmetic
-- | operations, and others.
data Expression = Negative Op    -- (-x)
  | Not Op                       -- (!x)
  | Equals Op1 Op2               -- (x == y)
  | NotEquals Op1 Op2            -- (x != y)
  | GreaterEquals Op1 Op2        -- (x >= y)
  | LessEquals Op1 Op2           -- (x <= y)
  | Greater Op1 Op2              -- (x > y)
  | Less Op1 Op2                 -- (x < y)
  | And Op1 Op2                  -- (x && y)
  | Or Op1 Op2                   -- (x || y)
  | BitwiseAND Op1 Op2           -- (x & y)
  | BitwiseOR Op1 Op2            -- (x | y)
  | BitwiseXOR Op1 Op2           -- (x ^ y)
  | LShift Op1 Op2               -- (x << y)
  | RShift Op1 Op2               -- (x >> y)
  | BitSelect Op1 Op2            -- (x[y])
  | BitSelectRange Op Op1 Op2    -- (x[y:z])
  | BitConcat [Op]               -- concat(x, y, z, ... )
  | Multiply Op1 Op2             -- (x * y)
  | Divide Op1 Op2               -- (x / y)
  | Modulo Op1 Op2               -- (x % y)
  | Add Op1 Op2                  -- (x + y)
  | Subtract Op1 Op2             -- (x - y)
  | Literal Lit                  -- (x)
  | Identifier ID_Path           -- (x.y)
  | Exp_MethodCall ID_Path MethodName [Expression]
    (Maybe Writes)               -- (mod.meth(x,y,z,...))
  | Exp_FunctionCall String [Expression]
                                 -- (func(x,y,z ,...) )
  | ValueMethodCall SuperModuleName ModuleName InstanceName
    MethodName                    -- (mod.meth()), includes annotations
  | Exp_If Guard Op1 Op2          -- (x)? y : z
  | Skip                          -- internal use
  | RPFlag Op                     -- internal use
  | MaybeIf Matching ID_Path Op1 Op2
                                  -- Pattern matching if statement for
                                     Maybe values
  | Tagged (Maybe PVSType) MaybeTag
                                  -- In-situ maybe wrapping
  | FromMaybe ID_Path Op          -- internal use
  | MaybeValue Op                 -- internal use
  | Binding [LocalVar] Expression-- Local Variable Binding
  | CasesOf Expression [ExpCase] -- Case expression
  | StructCluster (Either BSVType PVSType)
    [(String, Expression)]        -- Multiple fields of a structure
  | PMatch MaybeIDTag             -- Pattern match on maybes
  | IsValid MaybeTag              -- Maybe type test for validity
  | FieldAccess Op ID_Path        -- structure field access
deriving (Eq, Ord, Show)

type LocalVar = ( ID_Path
                , (Either (Maybe BSVType) (Maybe PVSType))
                , Expression )
type Matching = ID_Path
type Op = Expression
type Op1 = Expression
type Op2 = Expression
type ModuleName = String
type SuperModuleName = String
type InstanceName = String
type MethodName = String
type UTArgs = [(String, Maybe BSVType)]
    -- Arguments without type annotations
type StructName = String
type FieldName = String
```

END HASKELL

3.1.5 Statements

All `Statements` are either control flow or method invocation. Commonly used methods, such as register writes, may be used with syntactic sugar that gives the appearance of assignment. This is not to be confused with assignment in the imperative sense, which BSV does not support. The mapping between the types of statements presented in the grammar below is surjective, but not injective, as statements which are syntactically distinct are sometimes semantically equivalent, such as the three ways given for register write operations. In some cases these are straight syntactic sugar. In others, these are special cases for particular operations, such as if statements which perform maybe type extraction.

GRAMMAR

```
⟨Stmt⟩ ::= [ ⟨Stmt Pragma⟩ ] ⟨Statement⟩

⟨Statement⟩ ::= ⟨Identifier⟩ ‘<=’ ⟨Expression⟩ ‘;’
  |  ⟨Identifier⟩ ‘[’ ⟨Expression⟩ ‘] <=’ ⟨Expression⟩ ‘;’
  |  ⟨Identifier⟩ ‘._write(’ ⟨Expression⟩ ‘);’
  |  ⟨Identifier⟩‘.’⟨Identifier⟩[‘(’[⟨Expression⟩{‘, ’⟨Expression⟩}]‘)’]‘;’
  |  ⟨Identifier⟩ ‘;’
  |  ‘$’ ⟨string⟩ ‘;’
  |  ‘return’ ⟨Expression⟩ ‘;’
  |  ‘return’ ⟨BSV Type⟩ ‘{’ ⟨Record Stmt List⟩ ‘}’
  |  ‘if (’ ⟨Expression⟩ ‘)’ ⟨Stmt⟩ [ ‘else’ ⟨Stmt⟩ ]
  |  ‘if (’ ⟨Identifier⟩ ‘matches tagged Valid .’ ⟨Identifier⟩ ‘)’
     ⟨Stmt⟩
     [ ‘else’
     ⟨Stmt⟩ ]
  |  ‘for (’⟨Stmt⟩{‘, ’⟨Stmt⟩}‘;’⟨Expression⟩‘;’⟨Stmt⟩{‘, ’⟨Stmt⟩}‘)’
     ⟨Stmt⟩
  |  ‘case (’ ⟨Expression⟩ ’)’ [‘matches’]
     { ⟨case⟩ }
     ‘endcase’
  |  ‘let ’ ⟨Identifier⟩ ‘ <- ’ ⟨Expression⟩ ‘;’
  |  ⟨BSV Type⟩ ⟨Identifier⟩ ‘=’ ⟨Expression⟩ ‘;’
  |  ‘begin’ ⟨Stmt⟩ { ⟨Stmt⟩ } ‘end’
  |  ⟨Identifier⟩ ‘<=’ ⟨Name⟩ ‘{’ ⟨Record Stmt List⟩ ‘}’
```

⟨*Record Stmt List*⟩ ::= ⟨*Identifier*⟩‘`:`’⟨*Expression*⟩[‘`,`’⟨*Record Stmt List*⟩]
  | ⟨*Identifier*⟩ ‘`:`’ ⟨*Expression*⟩

⟨*case*⟩ ::= ⟨*Literal*⟩ { ‘`,`’ ⟨*Literal*⟩ } ‘`:`’ ⟨*Stmt*⟩

⟨*Stmt Pragma*⟩ ::= ‘`(*`’ ⟨*Stmt Att*⟩ { ‘`, `’ ⟨*Stmt Att*⟩ } ‘`*)`’

⟨*Stmt Att*⟩ ::= ‘`split`’
  | ‘`nosplit`’
  | ⟨*Common Att*⟩

⟨*Common Att*⟩ ::= ‘`descending_urgency = "`’ ⟨*string list*⟩ ‘`"`’
  | ‘`execution_order = "`’ ⟨*string list*⟩ ‘`"`’
  | ‘`mutually_exclusive = "`’ ⟨*string list*⟩ ‘`"`’
  | ‘`conflict_free = "`’ ⟨*string list*⟩ ‘`"`’
  | ‘`preempts = "`’ ⟨*string list*⟩ ‘`"`’
  | ‘`doc = "`’ ⟨*string*⟩ ‘`" `’

END GRAMMAR

Abstract Syntax in Haskell

As demonstrated below, statements can contain many sub-elements, including the names of actions and methods. In BSV, it is possible to attach attributes (or pragmas) at the individual statement level. Most attributes are not relevant to the translation process, particularly at the statement level, since scheduling pragmas are more likely to be included at the action or module levels. Nevertheless, scheduling pragmas are permitted to occur at this level, and must be accounted for, however out of place they may seem.

HASKELL

```
data Statement =
    Write ID_Path Expression [StatementAttribute]
     -- Register write operation
  | MethodCall ID_Path MethodName [Expression]
       [StatementAttribute]
     -- Action method invocation
  | ActionCall ActionName [StatementAttribute]
     -- Action block invocation
  | Return Expression [StatementAttribute]
     -- Return statement
  | StructReturn BSVType [(String, Expression)]
      [StatementAttribute]
    -- Return statement which returns a structure
  | If Guard Then Else [StatementAttribute]
    -- Garden Variety Conditional
  | PMatchIf ID_Path ID_Path Statement Statement
      [StatementAttribute]
    -- Pattern matching if statement for maybe types
  | ForLoop [Inits] Guard [Increments] Statement
      [StatementAttribute]
    -- For loops are only for syntactic iteration only
  | Switch Guard [Case] [StatementAttribute]
    -- Switch-case block
  | LocalDec [LocalVar] Statement [StatementAttribute]
    -- Local variable binding
  | StatementBlock [Statement]
    -- Statements grouped by 'begin' and 'end'
  | Void
    -- Internal use
  deriving (Eq)

type Case = (Literal, Statement)
type ExpCase = (Literal, Expression)
type Guard = Expression
type Then = Statement
type Else = Statement
```

END HASKELL

3.1.6 *Intra-module Interfaces*

When an interface is declared within a module, its syntax changes substantially enough to make it distinct from extra-module interface declarations. Intra-module interfaces encapsulate a number of method declarations as seen in 3.1.10, as opposed to method stubs. Methods inside intra-module interfaces are addressed through the declared interface name.

GRAMMAR

⟨*IntraModule Interface*⟩ ::= ‘`interface`’ ⟨*Name*⟩ ⟨*Name*⟩ ‘;’
{ ⟨*Method Stub*⟩ }
‘`end interface`’

END GRAMMAR

Running Example

In our running example, an interface is declared within the `mkTrafficLight` module. This permits additional points of access to be declared and instantiated within a module, accessed via dot syntax.

BSV

```
⋮

Reg#(Colour) lampState <- mkReg(Red);
Reg#(Int#(32)) T <- mkReg(0);

interface TrafficLight2;
  method Bool getYellow ();
  method Bool getRed ();
endinterface

Action reset =
  (action

⋮
```

END BSV

Abstract Syntax in Haskell

Intra-module interface declarations (or mid-module interface declarations) are simply a tagged collection of method stubs.

HASKELL

```
type MidModInterfaceDec = (String, String, [MethodBody])
```

END HASKELL

3.1.7 State Declaration

`State Declarations` instantiate submodules, whether they are custom defined, or a prelude module, such as `Register`. It should be noted that there are more types of `Vectors` and `FIFOs` than are presented here. Those extensions are beyond the scope of the present implementation. Implementation of these modes of operation is reserved as future work.

GRAMMAR

⟨*State Declaration*⟩ ::= ⟨*Name*⟩ ⟨*Identifier*⟩ ‘`<-`’⟨*Identifier*⟩‘`;`’
| ‘`Reg#(`’ ⟨*BSV Type*⟩ ‘`)`’ ⟨*Identifier*⟩ ‘`<- mkReg(`’ ⟨*Literal*⟩ ‘`);`’
| ‘`FIFO#(`’ ⟨*BSV Type*⟩ ‘`)`’ ⟨*Identifier*⟩ ‘`<- <FIFO Type>;`’
| ‘`Vector#(`’ ⟨*Literal*⟩ ‘`,`’ ⟨*Vector Type*⟩ ‘`)`’ ⟨*Identifier*⟩
‘`<-`’⟨*Vector Init*⟩‘`;`’
| ‘`RegFile#(`’ ⟨*BSV Type*⟩ ‘`,`’ ⟨*BSV Type*⟩ ‘`)`’ ⟨*Identifier*⟩
‘`<-`’⟨*Register Loader*⟩‘`;`’

⟨*FIFO Type*⟩ ::= ‘`mkSizedBypassFIFOF(`’ ⟨*Literal*⟩ ’)’
| ‘`mkSizedFIFO(`’ ⟨*Literal*⟩ ’)’
| ‘`mkSizedFIFOF(`’ ⟨*Literal*⟩ ’)’
| ‘`mkFIFOF`’
| ‘`mkFIFO`’

⟨*Vector Type*⟩ ::= ‘`Reg#(`’ ⟨*BSV Type*⟩ ‘`)`’

⟨*Vector Init*⟩ ::= ‘`replicateM(`’ ⟨*Constructor*⟩ ‘`)`’

⟨*Constructor*⟩ ::= ‘`mkReg(`’ ⟨*Expression*⟩ ‘`)`’

⟨*Register Loader*⟩ ::= ‘`mkRegFileLoad("`’⟨*Identifier*⟩‘`",`’
⟨*Literal*⟩‘`,`’⟨*Literal*⟩‘`)`’

END GRAMMAR

`Register Files` are registers with initialization data stored in files outside the source file. For more information regarding the various types of `FIFO` buffers, please see the BSV documentation Bluespec Inc. (2012a).

Running Example

The `mkTrafficLight` module declares two registers, `lampState` and `T`, representing the state of the traffic lamp and the time elapsed. One is of the enumerated `Colour` type, the other is a 32 bit integer.

——————————————— BSV ———————————————
```
                              ⋮

endinterface

module mkTrafficLight (TrafficLight);
  Reg#(Colour) lampState <- mkReg(Red);
  Reg#(Int#(32)) T <- mkReg(0);

  interface TrafficLight2;
    method Bool getYellow ();

                              ⋮
```
——————————————— END BSV ———————————————

Abstract Syntax in Haskell

State declarations are encoded as follows.

HASKELL
```
-- | State Declaration
data BSVstateDec = BSV_Reg ID_Path BSVType Init
   | BSV_FIFO FIFOType ID_Path BSVType
   | BSV_Vector ID_Path BSVType N VectorInit
   | BSV_RegFile ID_Path AddressWidth BSVType RegFileLoader
   | BSV_SubModuleDec InterfaceName Name InstName
   | DWire ID_Path BSVType Init
   deriving (Eq, Show)

-- | FIFOs may be initialized with any of the following
data FIFOType = FIFO | FIFOF | SizedFIFO Literal
  | SizedFIFOF Literal | DepthParamFIFO | FIFO1
  | FIFOF1 | LFIFO | LFIFOF | PipelineFIFO
  | PipelineFIFOF | BypassFIFO | BypassFIFOF
  | SizedBypassFIFOF Literal deriving (Eq, Show)

-- | Vectors may be initialized with the following
data VectorInit = Replicate Literal
  | Explicit [Literal] deriving (Eq, Show)
```

```
-- | Register files must have bounds specified
data RegFileLoader =
  RegFileLoad FileName MinIndex MaxIndex deriving(Eq,Show)

-- | The following are provided for the legibility .
type Name = String
type N = Integer
type Init = Expression
type InstName = String
type MinIndex = Literal
type MaxIndex = Literal
type AddressWidth = BSVType
```

END HASKELL

In other words, a state declaration may be a register, FIFO buffer, vector or submodule declaration. Each type option for the `BSVstateDec` type bundles together all of the information declared in the state declaration, such as the identifier of the declared element, any initialized value, et cetera. Submodule instantiations are included in this category because, from a semantic standpoint, the objects we consider state elements are in fact submodules. They are submodules supported by the Bluespec prelude, and have generous amounts of syntactic sugar supporting their use, but behaviourally they are submodules. One can even call `read` and `write` methods on a register in exactly the same way as a method of any other module. It just so happens that `read` is contextually implicit, and `write` can also be accomplished by means of the <= operator.

3.1.8 Action Declaration

An `Action` is a group of statements that may be declared separately from rules and methods, and invoked as a statement. Contrary to their name, `Actions` take no action by themselves, they are simply an abstraction mechanism for `Statement` blocks. Common uses for `Action` declarations are encapsulating frequently used or changed code, so that multiple instances of the same set of statements need not be modified individually each time a change is required.

GRAMMAR

⟨*Action Declaration*⟩ ::= [‘`(*`’ ⟨*Action Att*⟩ { ‘`,` ’ ⟨*Action Att*⟩ } ‘`*)`’]
    ‘`Action`’ ⟨*Identifier*⟩ ‘`=`’
    ‘`( action` ’
    ⟨*Stmt*⟩
    { ⟨*Stmt*⟩ }
    ‘`endaction );`’

⟨*Action Att*⟩ ::= ‘`doc = "`’ ⟨*string*⟩ ‘`"`’

END GRAMMAR

`Stmt` is defined in §3.1.5.

Running Example

The action given in the following code example is a simple reset routine, which merely resets the lamp state to `Green`.

BSV

```
                    ⋮

  method Bool getRed ();
endinterface

Action reset =
  (action
  lampState <= Green;
  endaction);

rule goYellow (lampState == Green && T >= 140);
  lampState <= Yellow;

                    ⋮
```

END BSV

Abstract Syntax in Haskell

Actions are encoded simply as a list of statements tupled with an identifier and a set of declared attributes.

HASKELL

```
type ActionDec = ( ActionName
                 , [Statement]
                 , [ActionAttribute]
                 )
```

END HASKELL

When invoked, the list of statements grouped together to form an action are inserted into the hardware description in the location of the action invocation.

3.1.9 Rule Declaration

`Rules` are actions which are eligible for scheduling by Bluespec's action arbitration mechanism. For an in-depth examination of the action arbitration semantic, please see §4.1. In both actions generally and rules specifically, the guard expression may be omitted. Semantically, this means that the rule is "always on". This can be modelled by using `True` as the guard expression. When a module performs calculations that are not directly controlled or initiated by a parent module, those calculations will typically be contained in a `Rule`.

GRAMMAR

⟨*Rule Declaration*⟩ ::= ['`(*`' ⟨*Rule Att*⟩ { '`, `' ⟨*Rule Att*⟩ } '`*)`']
  '`rule`' ⟨*Identifier*⟩ ['`(`' ⟨*Expression*⟩ '`)`'] '`;`'
  ⟨*Stmt*⟩
  { ⟨*Stmt*⟩ }
  '`endrule`'

⟨*Rule Att*⟩ ::= '`fire_when_enabled`'
 | '`no_implicit_conditions`'
 | '`descending_urgency = "`' ⟨*string list*⟩ '`"`'
 | '`execution_order = "`' ⟨*string list*⟩ '`"`'
 | '`mutually_exclusive = "`' ⟨*string list*⟩ '`"`'
 | '`conflict_free = "`' ⟨*string list*⟩ '`"`'
 | '`preempts = "`' ⟨*string list*⟩ '`"`'
 | '`doc = "`' ⟨*string*⟩ '`"`'

END GRAMMAR

`Return Type` is defined in §3.1.16.

The pragmas included at this level (`descending_urgency`, `conflict_free`, etc.) have direct scheduling implications, and are the only pragmas BAPIP currently takes into account.

In order to handle pattern matching over `Maybe` types, a pattern matching guard is transformed into a test for the `Maybe` having a value in the guard, and an encapsulation of all statements in a local definition for the specified identifier. While this introduces a slight difference between BSV and our abstract syntax, it reduces the complexity of the translation while having no impact on the underlying semantics. For example:

BSV

```
rule r (s matches tagged Valid q);
  <statements>
endrule

// Becomes

rule r (isValid s);
  let q = fromMaybe s in
    <statements>
endrule
```

END BSV

`Maybe` types are discussed in §3.1.1.

Running Example

The running example contains one rule, which causes the lamp state to change to yellow if it is first green, and if the timer value exceeds 139.

——————————————————— BSV ———————————————————

⋮

```
    lampState <= Green;
    endaction);

  rule goYellow (lampState == Green && T >= 140);
    lampState <= Yellow;
  endrule
```

```
method Action setTime(time);
                              ⋮
```

——————————— END BSV ———————————

Abstract Syntax in Haskell

Rules are actions with guards, or "guarded actions."

HASKELL

```haskell
type RuleDec = ( RuleName
               , Guard
               , [Statement]
               , [RuleAttribute]
               )
```

END HASKELL

At this level, we are not yet concerned with rule scheduling, just organizing the data contained in the BSV file. The next step is primarily concerned with rule scheduling.

3.1.10 Method Body Declaration

`Methods` are `Rules` which are not available for execution by default, but must be invoked by a parent module. Once invoked, `Methods` take the highest priority during action arbitration, but execution of a `Method` can still be blocked by that `Method`'s guard.

Method body declarations are distinct from the method stubs appearing in interface declarations (§3.1.16), in that method stubs merely declare the type of the method, including the types of its arguments and its return value, whereas a Method body declaration declares the statements a method executes and it's guard (if any).

GRAMMAR

⟨*Method Body Declaration*⟩ ::= [‘`(*`’⟨*Method Att*⟩{‘`, `’⟨*Method Att*⟩ }‘`*)`’]
‘`method`’ ⟨*Return Type*⟩ ⟨*Identifier*⟩ ‘`(`’ [⟨*Arglist*⟩] ‘`)`’
[‘`if (`’ ⟨*Expression*⟩ ‘`)`’] ‘`;`’

⟨*Stmt*⟩
{ '`;`' ⟨*Stmt*⟩ }
'`endmethod`'

⟨*Arglist*⟩ ::= ⟨*Argument*⟩ {'`,`' ⟨*Argument*⟩}

⟨*Method Att*⟩ ::= '`doc = "`' ⟨*string*⟩ '`"`'

END GRAMMAR

where `Return Type` and `Argument` is defined in §3.1.16.

Running Example

The `mkTrafficLight` module declares two interface methods, one for setting the time register, the other for outputting whether the lamp state is green or not.

BSV

```
                              ⋮

    endrule


    method Action setTime(time);
      T <= time;
    endmethod

    method Bool getGreen;
      return lampState == Green;
    endMethod

endmodule
                              ⋮
```

END BSV

Abstract Syntax in Haskell

Just as rules are actions with guards, methods are rules with arguments and a return type. In the case of input methods, the return type indicates whether the method is input, output, or both, which is why the Maybe monad has not been used.

HASKELL

```
type MethodBody = ( MethodName
  , ReturnType
  , UTArgs
  , Guard
  , [Statement]
  , [MethodBodyAttribute] )
type UTArgs = [(String, Maybe BSVType)]
```

END HASKELL

It should be noted that, while the arguments used in a method body declaration are recommended to be typed by the Bluespec compiler (i.e., a warning is tripped by the absence of typing information), this is not a strict requirement. Typing information is already provided by a module's interface declaration. There is also no strict requirement that the argument identifiers used in the method stubs in the interface declaration be reused in the method declaration. The identifiers used in the method's statements and guard must correspond to the identifiers declared in the method declarations.

3.1.11 Module Level Grammar

`Modules` are the highest level of hardware organization underneath packages, and are analogous to the modules of other Hardware Description Languages such as VHDL and Verilog. `Modules` declare an interface, which provides type information for all the methods provided by a module. When instantiated by parent modules, the interface of a `Module` is that `Module`'s *type*.

Intra-module interfaces are somewhat distinct from those declared outside of modules. These are discussed in §3.1.6.

GRAMMAR

$\langle$*BSV Module*$\rangle$ ::= ['`(*`' $\langle$*Module Att*$\rangle$ { '`, `' $\langle$*Module Att*$\rangle$ } '`*)`']
  '`module`' $\langle$*Identifier*$\rangle$ ' `(`' $\langle$*Identifier*$\rangle$ '`);`'
  { $\langle$*Module Level Declaration*$\rangle$ }
  '`endmodule`'

⟨*Module Level Declaration*⟩ ::= ⟨*State Declaration*⟩
  | ⟨*Action Declaration*⟩
  | ⟨*Rule Declaration*⟩
  | ⟨*IntraModule Interface*⟩
  | ⟨*Method Body Declaration*⟩

⟨*Module Att*⟩ ::= ‘`synthesize`’
  | ‘`always_ready = "`’ ⟨*Identifier*⟩ {‘`, `’ ⟨*Identifier*⟩} ‘`"`’
  | ‘`always_enabled = "`’ ⟨*Identifier*⟩{‘`, `’ ⟨*Identifier*⟩} ‘`"`’
  | ‘`clock_prefix = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`gate_prefix = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`reset_prefix = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`gate_input_clocks = "`’ ⟨*string*⟩{‘`, `’ ⟨*string*⟩} ‘`"`’
  | ‘`gate_all_clocks`’
  | ‘`default_clock_osc = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`default_clock_gate = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`default_gate_inhigh`’
  | ‘`default_gate_unused`’
  | ‘`default_reset = "`’ ⟨*string*⟩ ‘`"`’
  | ‘`no_default_reset`’
  | ‘`clock_family = "`’ ⟨*string*⟩{‘`, `’ ⟨*string*⟩} ‘`"`’
  | ‘`clock_ancestors = "`’ ⟨*string*⟩{‘`, `’ ⟨*string*⟩} ‘`"`’
  | ⟨*Common Att*⟩

END GRAMMAR

Running Example

This grammar encapsulates the `mkTrafficLight` module in its entirety. This module is a stripped-down version of that presented in §2.1, and is intended to demonstrate correct BSV syntax, with no attention paid to semantic content.

BSV

```
⋮

endinterface

  (*synthesize*)
module mkTrafficLight (TrafficLight);
  Reg#(Colour) lampState <- mkReg(Red);
```

```
    Reg#(Int#(32)) T <- mkReg(0);

    interface TrafficLight2;
      method Bool getYellow ();
      method Bool getRed ();
    endinterface

    Action reset =
      (action
      lampState <= Green;
      endaction);

    rule goYellow (lampState == Green && T >= 140);
      lampState <= Yellow;
    endrule


    method Action setTime(time);
      T <= time;
    endmethod

    method Bool getGreen;
      return lampState == Green;
    endMethod

  endmodule
```

⋮

——————————— END BSV ———————————

Abstract Syntax in Haskell

An observant reader will notice that in the previous definitions, use of record syntax corresponds to the use of permutation parsing. This is also the case with BSV modules, as demonstrated below.

HASKELL

```
data BSVModuleDec = BSVModuleDec
  { mName         :: String
  , instanceName  :: String
  , instances     :: [BSVModuleDec]
  , interfaceName :: String
  , interfaceDecs :: [MidModInterfaceDec]
  , attributes    :: [ModuleAttribute]
  , state         :: [BSVstateDec]
  , actions       :: [ActionDec]
  , rules         :: [RuleDec]
  , methods       :: [MethodBody]
  } deriving (Eq, Show)
```

END HASKELL

At this level, a distinction is drawn between the a module's name (`mName`), and its instance (`instanceName`), which must be considered in the context of the `instances` field. BSV modules instantiate submodules as part of their state declaration. The `instances` field contains a list of submodules declared by the module. This is not a reference to some globally defined module, it is an actual copy of the module declaration, which has had its `instanceName` field modified to the instance name assigned to it by its supermodule. The top-level module is given the instance name "root." This converts our encoding of a BSV description from a disjoint set of module declarations to a hierarchical structure of modules which may be traversed. This structure is left empty by the parser, and is later filled in during the scheduling preprocessor.

The `interfaceName` simply indicates which of the declared interfaces this module uses. The information contained therein describes the complete typing information for the methods used in this module. `interfaceDecs` however, is a list of all intra-module interface declarations made in the specified module.

3.1.12 Default Instance Declaration

`Default Instances` are a specific case of instance declaration in BSV, which is best likened to the type class instance in Haskell. When a structure is declared in BSV, it can become cumbersome to write a value out explicitly for each field in the structure, wherever a default value is called for. This is especially true given how large and frequently used structures can become. By importing the `DefaultValue` library in BSV, one may give structs an explicit default value. Then, whenever such a value is called for, `defaultValue` may be used in place of explicit declaration.

GRAMMAR

⟨*Default Instance Definition*⟩ ::= '`instance DefaultValue#(`'
  ⟨*Name*⟩ '`)` `{`' ⟨*Identifier*⟩ '`:`' ⟨*Expression*⟩
  {'`,`' ⟨*Identifier*⟩ '`:`' ⟨*Expression*⟩ }
  '`};`'

END GRAMMAR

where `Expression` is defined in §3.1.4.

Running Example

In this case, we define the default value of the struct `mystruct` such that all three elements are initialized to `False`.

BSV

```
                          ⋮

package TrafficLight;
      Bool c;
  } myStruct deriving (Eq);

  instance DefaultValue#(myStruct);
      defaultValue = myStruct a:False, b:False, c:False;
  endinstance

  interface TrafficLight;
    method Action setTime (Int#(32) time);

                          ⋮
```

END BSV

Abstract Syntax in Haskell

Instance definitions are collected for eventual substitution for struct constructor literals.

HASKELL

```haskell
type BSVInstDef = (Name, [(Name, Literal)])
```

END HASKELL

3.1.13 *Function Declaration*

Functions in BSV are roughly analogous to functions as defined in the functional programming paradigm. Given specific inputs, a function will produce the specified output, and have no side effects. In hardware terms, a function is a purely combinational circuit with no sequential elements.

GRAMMAR

```
⟨Function Definition⟩ ::= 'function' ⟨BSV Type⟩ ⟨Identifier⟩ ⟨Args List⟩
      ';'
      { ⟨Stmt⟩ }
      'endfunction'
```

END GRAMMAR

where `Arguments` are defined in §3.1.16, and `Statements` are defined in §3.1.5. While it may seem that the only `Statement` type allowable inside a function would be the `return` statement, such a statement may legally be preceded by many local variable declarations, or inside an if-else structure. Thus, it is necessary to read a collection of statements, rather than a single statement.

Running Example

The following function returns the increment of the 8 bit value passed, but returns the value itself if the number would overflow.

BSV

⋮

```
endmodule

function Bit#(8) safe_increment (Bit#(8) val) ;
  if (val == 8'hff)
    return 8'hff;
  else
```

```
        return val + 1;

  endpackage
```

———————————— END BSV ————————————

Abstract Syntax in Haskell

A function consists of a name, a return type, a list of arguments and a list of statements.

HASKELL

```
type BSVFunction = ( String
                   , [Argument]
                   , BSVType
                   , [Statement]
                   )
```

END HASKELL

3.1.14 Type Definition

`Type Definitions` may be used to define new types either by renaming existing types, via type enumerations, or using C-style `struct` constructors. While type classes are parsable, they are not utilized by the translation algorithm. Those type-classes relating to hardware production are simply ignored, as PVS deals with these values as enumerations themselves. Some type-classes are not addressed, and perhaps could be in future work, such as `Ord`, `Bounded` and `Arith`. The others are assumed to hold, even if not declared. Because the BSV designs are assumed to be compilable by the Bluespec compiler prior to translation, the original description cannot use any constructs not in conformity with declared type-classes. Thus, if `Eq` is not declared, == could not be used on the enumeration type. It does not alter the semantics of the description to attribute more type-classes than are declared if they are never used.

.... GRAMMAR ....

⟨*Type Definition*⟩ ::= ‘`typedef`’ ⟨*BSV Type*⟩ ⟨*Name*⟩ ‘`;`’
  | ‘`typedef enum {`’ ⟨*Identifier*⟩ {‘`, `’ ⟨*Identifier*⟩ } ‘`}`’ ⟨*Name*⟩
    [‘`deriving (`’ ⟨*Type Class*⟩ { ‘`, `’ ⟨*Type Class*⟩ } ‘`)`’]

| 'typedef struct {' [⟨*Fields*⟩] '}' ⟨*Name*⟩
['deriving (' ⟨*Type Class*⟩ { ', ' ⟨*Type Class*⟩ } ')']

⟨*Type Class*⟩ ::= 'Bits' | 'Eq' | 'Literal' | 'Ord' | 'Bounded' | 'Bitwise'
| 'BitReduction' | 'BitExtend' | 'Arith'

⟨*Fields*⟩ ::= ⟨*BSV Type*⟩ ⟨*Identifier*⟩ ';' ⟨*Fields*⟩
| ⟨*BSV Type*⟩ ⟨*Identifier*⟩ ';'

END GRAMMAR

Some types are parameterized by bit-width. Using typedefs, we can declare types of fixed width, which otherwise behave like their carrier type.

Running Example

Here we see typedefs of all three flavours. An enumeration gives us more human-readable lamp states. A regular type def identifies 50 bit registers as addresses. Finally, we create a structure containing three Booleans.

BSV

```
⋮

String client1_req_msg = "Client 1 requesting.";

typedef enum Green, Yellow, Red   Colour;
typedef Bit#(50)  Addr;

typedef struct {Bool a;
    Bool b;
    Bool c;
} myStruct deriving (Eq);

instance DefaultValue#(myStruct);
    defaultValue = myStruct a:False, b:False, c:False;

⋮
```

END BSV

Abstract Syntax in Haskell

Type definitions are defined in Haskell as follows.

HASKELL

```
data BSVTypeDef = BSV_Synonym Name BSVType
| BSV_Enumeration Name [Enumerat]
| BSV_Struct Name [BSV_Field] deriving (Eq, Show)
type Enumerat = String
type BSV_Field = (Name, BSVType)
```

END HASKELL

A type definition can be either a type synonym or an enumeration. Both require a name, but the type synonym requires the type that the new type identifier will be a synonym of, and the enumeration requires a list of enumerats, which are simply strings.

3.1.15 Constant Declaration

`Constant` values may be declared in BSV programs in order to utilize commonly used literal values symbolically, either for convenience or information hiding.

GRAMMAR

$\langle$*Constant Declaration*$\rangle$ ::= $\langle$*BSV Type*$\rangle$ $\langle$*Identifier*$\rangle$ '`=`' $\langle$*Literal*$\rangle$ '`;`'

END GRAMMAR

Running Example

Although strings can not be generated to hardware in BSV, they can still be used for print statements used by the BSV compiler for the purposes of debugging. This constant declaration encodes a debug message.

BSV

```
                              ⋮

import FIFO ::*;

String client1_req_msg = "Client 1 requesting.";

typedef enum Green, Yellow, Red   Colour;
typedef Bit#(50)  Addr;

                              ⋮
```

END BSV

Abstract Syntax in Haskell

Constant declarations are easily defined in Haskell.

HASKELL

```
type BSVConstantDec = (Name, BSVType, Literal)
```

END HASKELL

3.1.16 Interface Declaration

Interface declarations abstract the declaration of a module's interface methods from the module itself, allowing multiple modules to share the same interface. A module will declare which interface it is using in its own declaration. The interface is where all methods potentially possessed by a module must have their typing information fully specified.

The "Attribute Lists" appearing below are pragmas that may or may not be specified by the user. While the set of pragmas being parsed in these contexts is comprehensive, semantically they are not taken into consideration by the translation algorithm, as most relate to hardware details that are unnecessary at this level of abstraction. For example, the `synthesize` pragma, placed at the beginning of a module, specifies that said module is to be synthesized into hardware. This has no impact on the logic we are analyzing, and is therefore not relevant enough to keep.

It is also possible in BSV to include other interfaces declared elsewhere, even in other packages, in an interface declaration. This adds the specified methods to any module invoking the interface, but those methods must be addressed through the local name given that interface within the super interface.

GRAMMAR

⟨*Interface Declaration*⟩ ::= [‘`(*`’⟨*Interface Att*⟩{‘`, `’⟨*Interface Att*⟩}‘`*)`’]
  ‘`interface`’ ⟨*Identifier*⟩ ‘`;`’
  { ⟨*Method Stub*⟩ }
  ‘`end interface`’

⟨*Method Stub*⟩ ::= ‘`method`’ ⟨*Return Type*⟩ ⟨*Identifier*⟩ ⟨*Args List*⟩ ‘`;`’
 | ‘`interface`’ ⟨*Name*⟩ ⟨*Name*⟩

⟨*Return Type*⟩ ::= ‘`Action`’
 | ‘`ActionValue #(`’ ⟨*BSV Type*⟩ ‘`)`’
 | ⟨*BSV Type*⟩

⟨*Args List*⟩ ::= ‘`(`’ ⟨*Argument*⟩ { ‘`, `’ ⟨*Argument*⟩ } ‘`)`’ | ‘`()`’

⟨*Argument*⟩ ::= [‘`(*`’ ⟨*Argument Att*⟩ { ‘`, `’ ⟨*Argument Att*⟩ } ‘`*)`’]
  ⟨*BSV Type*⟩ ⟨*Identifier*⟩

⟨*Interface Att*⟩ ::= ‘`always_ready`’
 | ‘`always_enabled`’
 | ‘`doc = "`’ ⟨*string*⟩ ‘`"`’

⟨*Argument Att*⟩ ::= ‘`ready = "`’ ⟨*Identifier*⟩ ‘`"`’
 | ‘`enable = "`’ ⟨*Identifier*⟩‘`"`’
 | ‘`result = "`’ ⟨*Identifier*⟩ ‘`"`’
 | ‘`prefix = "`’ ⟨*Identifier*⟩ ‘`"`’
 | ‘`port = "`’ ⟨*Identifier*⟩ ‘`"`’
 | ‘`always_ready`’
 | ‘`always_enabled`’
 | ‘`doc = "`’ ⟨*string*⟩ ‘`"`’

END GRAMMAR

Running Example

Here we see method stubs of both methods declared in our module. A method stub functions analogously to a function prototype in C, providing type information for a module's (or function's) interface.

——————————————————— BSV ———————————————————

```
⋮

    defaultValue = myStruct a:False, b:False, c:False;
endinstance

interface TrafficLight;
  method Action setTime (Int#(32) time);
  method Bool getGreen ();
endinterface

  (*synthesize*)

⋮
```

————————————————— END BSV —————————————————

Abstract Syntax in Haskell

The definition of the `InterfaceDec` type is as follows.

HASKELL

```
type InterfaceDec = ( Name
                    , [MethodDec]
                    , [InterfaceRef]
                    , [InterfaceAttribute]
                    )
type MethodDec = ( Name
                 , ReturnType
                 , [Argument]
                 , [MethodDecAttribute]
                 )
data ReturnType = Action
  | ActionValue BSVType
  | Value BSVType deriving (Eq, Show)
type Argument = (Name, BSVType, [ArgumentAttribute])
type InterfaceRef = (InterfaceName, String)
type InterfaceName = String
```

END HASKELL

Each interface declaration consists of a name and a list of method declarations, as distinct from method body declarations occurring within modules. Method declarations consist of a name, a return type, and a list of arguments. Valid return types include `Action`, `ActionValue`, and `Value`. Both `ActionValue`, and `Value` additionally have a BSVType specified, which signifies the type of the return value of the method

3.1.17 Import Declaration

`Import` statements allow a BSV package to include other BSV packages as libraries, in the same manner as an `include` statement. A BSV package may include any number of `Import` statements. While it is possible to import specified modules from a package via `Import` statements by naming them in the place of the asterisk below, at the semantic level the lazy evaluation of the translation will only use modules which are used by the top-level module's module hierarchy, even though all modules in every imported package are parsed.

GRAMMAR

$\langle$*Import Declaration*$\rangle$ ::= '`import`' $\langle$*Identifier*$\rangle$ '`:: *;`'

END GRAMMAR

Running Example

In this case, we are importing the FIFO module. Now FIFOs may be used in this package.

BSV

```
package TrafficLight;

  `include "RapidIO.defines"

  import FIFO ::*;

  String client1_req_msg = "Client 1 requesting.";
```

⋮

END BSV

Abstract Syntax in Haskell

In our Haskell abstract syntax data structure, import declarations are stored simply as a list of imported package names.

3.1.18 Include Declaration

`Include` statements are similar to `Import` statements, but are used specifically to include files with a `.define` extension, which contain macros. macros are discussed in §3.1.

GRAMMAR

⟨*Include Declaration*⟩ ::= '`` `include "``' ⟨*Identifier*⟩ '`"`'

END GRAMMAR

While macro substitutions are performed as a preprocessing step, include statements are never removed from the files. They are instead parsed as a separate syntactic construct and ignored.

Running Example

This define statement indicates that all the macros contained in the file `RapidIO.defines` should be applied to the source code in this package, and all packages which import this package.

BSV

```
package TrafficLight;

  `include "RapidIO.defines"

  import FIFO ::*;

                              ⋮
```

END BSV

Abstract Syntax in Haskell

Similarly to import declarations, include statements are stored simply as a list of the filenames of the files imported.

3.1.19 Package Level Grammar

The top-level entity of a BSV file is the `Package`, typically the container for the entire file. A `Package`, and the entities contained at this level, are defined by the following grammar productions:

GRAMMAR

⟨*BSV Package*⟩ ::= ‘`package`’ ⟨*Identifier*⟩ ‘`;`’
  { ⟨*Package Level Declaration*⟩ }
  ‘`endpackage`’

⟨*Package Level Declaration*⟩ ::= ⟨*Import Declaration*⟩
  | ⟨*Include Declaration*⟩
  | ⟨*Interface Declaration*⟩
  | ⟨*Constant Declaration*⟩
  | ⟨*Type Definition*⟩
  | ⟨*Function Declaration*⟩
  | ⟨*Default Instance Declaration*⟩
  | ⟨*BSV Module*⟩

END GRAMMAR

Please note that the grammatical definition of a BSV `Module` will be addressed in §3.1.11.

Running Example

The package level grammar and its subordinates encompass the entirety of a BSV file.

BSV

```
package TrafficLight;

  `include "RapidIO.defines"

  import FIFO ::*;

  String client1_req_msg = "Client 1 requesting.";

  typedef enum Green, Yellow, Red   Colour;
  typedef Bit#(50)  Addr;
  typedef struct {Bool a;
      Bool b;
      Bool c;
  } myStruct deriving (Eq);

  instance DefaultValue#(myStruct);
      defaultValue = myStruct a:False, b:False, c:False;
  endinstance

  interface TrafficLight;
    method Action setTime (Int#(32) time);
    method Bool getGreen ();
  endinterface

    (*synthesize*)
  module mkTrafficLight (TrafficLight);
    Reg#(Colour) lampState <- mkReg(Red);
    Reg#(Int#(32)) T <- mkReg(0);

    interface TrafficLight2;
      method Bool getYellow ();
      method Bool getRed ();
    endinterface

    Action reset =
      (action
      lampState <= Green;
      endaction);

    rule goYellow (lampState == Green && T >= 140);
      lampState <= Yellow;
    endrule


    method Action setTime(time);
      T <= time;
    endmethod
```

```
      method Bool getGreen;
        return lampState == Green;
      endMethod

    endmodule

    function Bit#(8) safe_increment (Bit#(8) val) ;
      if (val == 8'hff)
        return 8'hff;
      else
        return val + 1;

  endpackage
```

——————————————— END BSV ———————————————

Abstract Syntax in Haskell

At the top level, a BSV package is encoded as a record, with fields as shown below.

HASKELL

```
data BSVPackage = BSVPackage
  { bsv_packageName :: PackageName
      -- the name of this package
  , imports         :: [PackageName]
      -- packages this package imports
  , including       :: [String]
      -- invoked .define files
  , interfaces      :: [InterfaceDec]
      -- declared module interfaces
  , bsv_constants   :: [BSVConstantDec]
      -- declared constants
  , bsv_typedefs    :: [BSVTypeDef]
      -- declared type definitions
  , bsv_instDefs    :: [BSVInstDef]
      -- structure default value defs
  , bsv_modules     :: [BSVModuleDec]
      -- declared modules
  , bsv_functions   ::  [BSVFunction]
      -- declared functions
  , bsv_macros      ::  [BSVMacro]
      -- macro substitutions
  , hexFiles        ::  [HexFile]
      -- Contents of files loaded into RegFile elements.
  } deriving (Show)

type BSVFunction = ( String
                   , [Argument]
                   , BSVType
                   , [Statement]
                   )
type BSVInstDef = (Name, [(Name, Literal)])
data BSVMacro = SimpleMacro String String
| CompoundMacro String [String] String deriving (Show,Eq)
type HexFile = ( String , [Literal] )
type Argument = (Name, BSVType, [ArgumentAttribute])
```

END HASKELL

In this context, `PackageName` is a synonym for the `String` type. In BSV, one may declare register files, which are, for all intents and purposes, vectors initialized by the values stored in files in hexadecimal format.

3.2 Data Structures Supporting an Abstract Syntax for a PVS Embedding

Just as we encode the abstract syntax of BSV in Haskell datatypes, so do we generate a similar representation of our eventual PVS product. Not only does this afford us the same benefits of type-correctness, but it affords us the opportunity to demonstrate the close correspondence between many parts of BSV and PVS. We will be comparing and contrasting the corresponding BSV structures throughout this section, but more algorithmically complex mappings are left for §3.6. For a list of all files generated by BAPIP, see §A.1.1.

Design Goals and Decisions, and Coverage of PVS

The design goal of this set of data structures was to support the present translation effort, and was in no way meant to be a general or complete encoding of PVS itself. The only PVS language constructs addressed are those directly touching on the PVS implementation of the logical model extracted at previous stages of the translator. Most of the language constructs discussed in this section in fact encode the design decisions of the BAPIP translation process.

Because this is not a general abstract syntax for PVS, its utility to other projects as such would be severely restricted. Thus, this abstract syntax has not been contributed to hackage.

3.2.1 PVS Package

The moniker "Package" is not derived from any endemic property of the data structure containing the translated PVS information, but rather because it is the corresponding symmetric object to the BSV package.

HASKELL

```
data PVSPackage = PVSPackage
  { pvs_packageName  :: PackageName
  , pvs_constants    :: [PVSConstantDec]
  , pvs_typedefs     :: [PVSTypeDef]
  , transitions      :: PVStransitions
  , pvs_state        :: [PVSstateDec]
  , pvs_instantiations :: [PVSInstDef]
  , pvs_functions    :: [PVSFunction]
  } deriving (Show)
```

END HASKELL

The lesser degree of complexity in the PVS package is readily apparent when compared to the BSV package. The main reasons for this are that all the imports have been flattened into this one structure, and interface information is discarded as unnecessary. Otherwise, the main difference is the absence of a structure symmetrical to the BSV module.

3.2.2 PVS Constant Declaration and Type Definition

Given the similar natures of type definitions and constant declarations in both languages, translating from one to the other is mostly a matter of changing values of type `BSVType` to `PVSType`, which is also nearly trivial.

HASKELL

```
data PVSTypeDef = PVS_Synonym Name PVSType
  | PVS_Enumeration Name [Enumerat]
  | PVS_Struct Name [PVS_Field]
  deriving (Eq, Show)

type PVS_Field = (Name, PVSType)

type PVSConstantDec = (Name, PVSType, Literal)
```

END HASKELL

3.2.3 PVS State Declaration

A PVS state declaration is a list of sub-declarations packaged with the name of the module they declare the state of. In BSV, the actual structure of the module hierarchy is set by the manner in which modules invoke each other as submodules *within the modules' state declarations*. We opt to express this as a list of module state declarations, which contain lists of individual state declarations, some of which may be references to other declared submodules. Similarly to type declarations, this information is extracted from a BSV module record via simple traversal.

HASKELL

```haskell
type PVSstateDec = (Name, [PVSstate])

data PVSstate = PVS_Reg ID_Path PVSType Init
  | PVS_FIFO FIFOType ID_Path PVSType
  | PVS_Vector ID_Path PVSType N VectorInit
  | PVS_SubModuleDec InterfaceName Name InstName
  | PVS_DWire ID_Path PVSType Init
  deriving (Eq, Show)
```

END HASKELL

Some primitive data types are shared between BSV and PVS records. These include `VectorInit` and `FIFOType` (see §3.1.7).

3.2.4 PVS Transitions

The transition file in our PVS output is pivotal to the modelling of complex behaviour. The process of producing transitions is the most algorithmically complex element of the overall translation. However, there are some peripheral elements to the production of the actual transition predicates which we will address here. A PVS transition corresponds to one user-specified schedule.

HASKELL

```haskell
type PVStransition =
    ( Integer
    , [(MethodName, [(MethodArg, PVSType)])]
    , [ValueMethod]
    , [TransitionTable]
    )
```

END HASKELL

A PVS transition is a tuple containing an index indicating the schedule number, a list of the input methods invoked in that schedule and their arguments, a list of value method definitions, and a transition table for all state elements.

Value Methods

Value methods are a type of method that may be declared in a BSV file, but have no effect on the state of the file when invoked. That said, a value method may return complex expressions, not just individual register values, and those values may use wires. Since wires are often set by method invocation, the output value of a value method must take into account the entirety of the schedule.

HASKELL

```haskell
type ValueMethod =
  ( MethodName
  , ModuleName
  , String
  , Path
  , PVSType
  , Expression
  , [ID_Path]
  )

type Path = [String]
```

END HASKELL

Transition Tables

The module transition type is a container for all of the information required to generate transition predicates, which is more that just simply the transition table. It is also necessary to know which, if any, methods have been included in the transition currently under consideration, and it is necessary to know what, if any, arguments are required by these top-level method invocations.

HASKELL

```haskell
data TransitionTable = TransMod Name [TransitionTable]
  | TransReg ID_Path SpecificTree
  | TransVect ID_Path Size [(Expression, SpecificTree)]
  | TransStruct Name [TransitionTable]
  | TransDWire ID_Path SpecificTree DefaultValue
  | TransFIFO ID_Path EnqTree DeqTree ClearTree
  deriving (Show, Eq)

type EnqTree = SpecificTree
type DeqTree = SpecificTree
type ClearTree = SpecificTree
```

END HASKELL

Transition tables correspond to a record update over the state record type. Each element of the state record has an entry in the transition table, and submodules are represented by including a nested transition table. Elements such as registers which have values modified do so by means of the "transition tree," discussed in §3.2.5

FIFO entries contain three trees, one for enqueueing operations, one for dequeueing operations, and one for clearing operations. Since these operations must occur in a specific order, the default value for dequeue is the result of enqueue, and the default of clear is the value of dequeue. These are all handled as local variables in PVS, with the value at the end of the execution of the clear tree (regardless whether it contains anything or not) is written into memory.

3.2.5 Transition Trees

Transition trees are binary trees which correspond to branching `if` statements. Each if statement, instead of including an expression to denote the condition, includes a reference to the relevant rule guard, as each branch of the transition tree represents the decision the action arbitrator makes about whether each rule fires or not. Transition trees terminate with "Leaf" expressions, which encode the expression which gets written to the register in the particular case represented by its position in the transition tree.

Transition trees pass through an intermediate form before reaching the final "Specific Tree" form. The "Total Tree," whose generation is described in detail in §4.2, is as follows. Note how each node in the tree has a list of statements and a guard. This is because each node in the total tree corresponds to an action in a BSV module.

HASKELL

```haskell
data TotalTree
  = TotalStem Guard [Statement] TrueTree FalseTree
  | TotalLeaf Guard [Statement]
  deriving (Eq)

type TrueTree = TotalTree
type FalseTree = TotalTree
```

END HASKELL

The “Specific Tree” is a traversal of the total tree, keeping only those statements relevant to the state entry at hand.

HASKELL

```haskell
data SpecificTree
  = SpecStem Guard (Either Expression SpecTrueTree) SpecFalseTree
  | SpecLeaf Guard TrueExpression FalseExpression
  | SpecEx Expression
  deriving (Eq)

type SpecTrueTree = SpecificTree
type SpecFalseTree = SpecificTree
type TrueExpression = Expression
type FalseExpression = Expression
```

END HASKELL

3.2.6 Functions

The conversion from a BSV function to a PVS function is extremely straightforward, comprised mostly in converting from BSV types to PVS types in the return type and in the arguments. Unlike methods, functions really do have no interest in state or scheduling, as they cannot contain identifiers that are not declared arguments.

HASKELL

```haskell
type PVSFunction = ( String
                   , [PVSArgument]
                   , PVSType
                   , Expression
                   )
type PVSArgument = (String, PVSType)
```

END HASKELL

BSV functions are discussed in §3.1.19.

3.2.7 Instance Definitions

PVS instance definitions are identical to the BSV, referenced in §3.1.12.

HASKELL

```haskell
type PVSInstDef = (Name, [(Name, Literal)])
```

END HASKELL

3.3 A Grammar for PVS Output Files

After finding intermediate representation in an abstract syntax tree, the design under examination then passes into concrete PVS syntax via generation. While it is possible to create a grammar of PVS output, and at points in this project this was accomplished, the additions and modifications made to the PVS grammar over the course of the last few years of the project have increased the complexity of this grammar to such a point that producing a full grammar is no longer a straightforward matter of organizing data contained in the source code of the generator.

It is due to the complexity of this grammar, as well as the multitude of design decisions embedded in BAPIP, that BSV files are not recoverable from PVS files by means of reverse-translation. While this avenue was explored earlier on in the project, it became less and less viable a feature as BSV2PVS increased the complexity to accommodate real-world examples. Enforcing this constraint would have forced us to accept a much smaller subset of BSV, and would have even had a negative effect on the freedom to implement in

PVS such things as wire handling and the action arbitration mechanism. As such, the prospect of a reversible translation is left as future work.

If such a reverse translation existed, it would be reasonably straightforward to construct a grammar from the parser. It would also be reasonably straightforward to construct a parser, given a grammar of BAPIP generated PVS code.

3.4 A Treatment of Tabular Specifications

Tabular specifications are a rigorous and formal way to specify the behaviour of software. These specifications feature prominently in the work of (Pang et al., 2015), which the present work is in part an extension of. Since BAPIP includes a mode for the generation of BSV code directly from such tabular specifications, it therefore follows that we must define a grammar for them.

The set of valid `.tsp` files used as input for BAPIP's `TSP2BSV` and `TSP2PVS` modes are a proper subset of the PVS specification language, so all `.tsp` files are valid, typecheckable PVS files. That said, the grammar only of the subset of PVS which constitute valid `.tsp` files, from the perspective of parsing and generating by BAPIP. While the tabular expressions encoded in `.tsp` files are in fact also valid PVS code, and can be used in other contexts for verification purposes, the PVS Tabular Specification file format (TSP) parser *does not* function as a general parser of PVS files.

3.4.1 Tabular Specification Grammar

In this section we will specify a grammar for the tabular specification files accepted by the `TSP2BSV` and `TSP2PVS` modes of BAPIP. This grammar will be expressed in extended Backus Naur form (EBNF) §2.5. Grammatical expressions for basic constructs such as expressions, types, and identifiers are shared with the BSV grammar, and may be found in §3.1.

GRAMMAR

⟨*TSP File*⟩ ::= ⟨*Header*⟩ {⟨*Import*⟩} {⟨*Content*⟩} ⟨*Footer*⟩

⟨*Header*⟩ ::= ⟨*Theory Declaration*⟩ ⟨*Begin*⟩

⟨*Theory Declaration*⟩ ::= ⟨*Name*⟩
  ['`[(IMPORTING Time) delta_t:posreal]`'] '`: Theory`'

$\langle$*Begin*$\rangle$ ::= '`BEGIN`' | '`begin`'

$\langle$*Import*$\rangle$ ::= '`IMPORTING`' $\langle$*Theory Reference*$\rangle$

$\langle$*Theory Reference*$\rangle$ ::= $\langle$*Name*$\rangle$ ['`[`' $\langle$*Identifier List*$\rangle$ '`]`'] ['`@`' $\langle$*Name*$\rangle$]

$\langle$*Identifier List*$\rangle$ ::= $\langle$*Identifier*$\rangle$ {'`, `' $\langle$*Identifier List*$\rangle$}

$\langle$*Content*$\rangle$ ::= $\langle$*Variable Declaration*$\rangle$
 | $\langle$*Table Declaration*$\rangle$

$\langle$*Variable Declaration*$\rangle$ ::= $\langle$*Identifier List*$\rangle$ '`: VAR`' $\langle$*Type*$\rangle$

$\langle$*Table Declaration*$\rangle$ ::= $\langle$*Name*$\rangle$ '`(`' $\langle$*Identifier List*$\rangle$ '`)(t) :`' $\langle$*Type*$\rangle$ '`=`'
 $\langle$*Table Assignment*$\rangle$
 $\langle$*Table*$\rangle$

$\langle$*Table Assignment*$\rangle$ ::= $\langle$*Identifier*$\rangle$ $\langle$*Time Specification*$\rangle$ '`=`'

$\langle$*Time Specification*$\rangle$ ::= '`(next(t))`' | '`(t)`'

$\langle$*Table*$\rangle$ ::= '`TABLE`'
 $\langle$*Table Line*$\rangle$
 {$\langle$*Table Line*$\rangle$}
 '`END TABLE`'

$\langle$*Table Line*$\rangle$ ::= '|' $\langle$*Boolean Expression*$\rangle$ '|' $\langle$*Expression*$\rangle$ '||'

$\langle$*Footer*$\rangle$ ::= '`End`' $\langle$*Name*$\rangle$

END GRAMMAR

In general, a TSP file is a PVS theory containing one or more tables, which make use of the `Time` libraries by Pang et al. (2015). It should be noted that the grammar for TSP file parsing is far less complex than that of a BSV file. This should give some indication of the range of expressivity of these two grammars.

3.4.2 Tabular Specification Abstract Syntax

Post parsing, the tabular specification is organized into a `TSPpackage` data structure. The structure is organized as follows.

HASKELL

```
data TSPpackage = TSPpackage {
    tName     :: String
  , typedefs  :: [PVSTypeDef]
  , defInsts  :: [PVSInstDef]
  , varDecs   :: [TVarDec]
  , tsps      :: [TSPTable]
  , tsp_funcs :: [PVSFunction]
  , macros    :: [PVSMacro]
  } deriving (Show, Eq)

type TVarDec = ([String], PVSType)

type PVSMacro = (String, PVSType, Literal)
```

END HASKELL

Type definitions, functions, Macros and instance definitions are used as per their definitions in §3.2. The main elements contained in this data structure are variable declarations, and the tabular expressions themselves. Variable declarations are encoded simply as all of the declared identifiers tupled to their declared type. These variables are not used to generate some sort of symmetric structure in BSV, but are used to fill in typing information.

HASKELL

```
type TSPTable = (TName -- Table name
  , TSPOutVar          -- output variable
  , Expression         -- init expression
  , [Replacement]      -- variable synonyms
  , [String]           -- input variables
  , [TSPLine]          -- table lines
  )

type TSPLine = (Guard, Expression)
type TSPOutVar = (ID_Path, Temporal)
data Temporal = N_Time Int deriving (Eq, Show)
type TName = String
```

END HASKELL

Tables contain a non-trivial amount of miscellany, such as declared local variables, input variable identifiers, initialization expressions, among them. All this is framing for the individual tabular specification lines, which are composed simply of a Boolean guard and a resulting expression.

3.4.3 A Note on Generation

Once the design has been fully parsed, and the relevant data structures generated, the design is always converted to BSV, regardless as to whether we are in `TSP2BSV` or `TSP2PVS` mode. For a grammar of the BSV output, see the BSV input grammar presented in §3.1. For notes on why a corresponding grammar of PVS does not appear in this thesis, see §3.3

3.5 Conclusion

The grammar specified in this chapter sets firm limits on what is and is not covered by the BAPIP translation software. Creation of the glsBAPIP software tool would have been impossible without such a grammar, implicit or explicit. Between our understanding of the grammar of our BSV subset, and the algorithm used to translate said subset presented in §4, we are now ready to discuss how one would go about using the product of this set of syntax and semantics to go about creating proof sequents, and proving them.

4. DERIVING STATE INTERACTIONS FROM BSV ACTION ARBITRATION SEMANTICS

One of the primary algorithmic contributions of this work is a conversion algorithm, taking the action-centred semantics of BSV and producing the state-centred semantics of a Kripke structure. This chapter details this process, provides illustration, discusses limitations of the process, and how those limitations were overcome.

4.1 BSV Semantics

Before the algorithm is described, it is important to have a clear understanding of the underlying semantics of BSV's guarded actions. Each action is comprised of a combinational circuit, which sends its results to a state-holding element, normally a register. In situations where multiple circuits may write their results to the same register, these data wires must be multiplexed so that only one may write to a register on any given clock cycle. A very basic multiplexer is shown in Figure 4.1.

Multiplexers are very common electronic components used in register-transfer level electronic designs. A multiplexer receives a selector signal, and

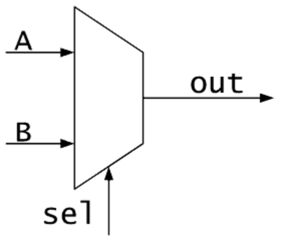


Figure 4.1: A 2x1 Multiplexer

based on that signal, outputs exactly one of its input signals. A selector signal with a bit-width of n can switch between 2^n input signals. In BSV, a multiplexer of the style given in Figure 4.1 would be given by the expression `if sel then A else B`. The wire `out` in the diagram is equivalent to the evaluated value of the above expression, and would simply be "plugged in" to the next step of the calculation.

The question this section answers is, while the hardware described by a BSV design is performing calculations, how are the selector signals of these multiplexers generated? The answer is the action arbitration algorithm. This algorithm is implemented as a control circuit, and executes concurrently with the rest of the hardware's calculations, controlling those calculations as state changes, inputs are received, and outputs are requested. This control circuit is necessarily specific to the hardware being designed, and is generated at compile time.

We will begin with a mathematical overview of the algorithm, and then give a full example in §4.2.

4.1.1 The BSV Transition System

The use of finite state machines for hardware modelling is standard practice for single-purpose processor design (Davis and Reese, 2008). To model BSV designs in PVS, we use the Kripke structure (Bowen, 1979), a similar state transition system. Kripke structures were used previously by Richards and Lester (2011) to model BSV in PVS, and, as this work derives from Richards' work, the same fundamental abstraction is used. We model a BSV hardware module as:

$$K = (S, s_0, \rightarrow, L)$$

In hardware, state is held in sequential components, or those containing flip-flops. S is the set of all possible state permutations. If R is the set of all sequential components defined in our target module, and $B = \{1, 0\}$ is the set of the valid values of any particular flip-flop, we can then say that the set of possible values of a state element ranges over $B \times B \times \cdots \times B$, where the number of B terms is the bit-size of the state element. S is defined more formally as the Cartesian product of the value ranges of each state element.

In BSV, sequential hardware elements must have a concrete initialization value. The permutation of bits stored in memory composed by these initializations, $s_o \in S$, is the initial state.

The transition relation $\rightarrow$ relates S back to itself, or $\rightarrow \subseteq S \times S$. It is left-total, so that $domain(\rightarrow) = S$. Implicitly, state elements retain their previously held value unless modified. This is interpreted as reflexivity, and produces left-totality for all states not explicitly addressed by module actions.

The labelling function L describes the mapping of elements of S into physical register values.

The purpose of this discussion is to document the properties of the transition relation $\rightarrow$, but over the course of this discussion, we will refine the above model. In addition to the above, we will make use of A, the set of actions a module defines, which includes both rules and methods.

4.1.2 Arbitration

The goal of the arbitration algorithm is to determine, from the set of all actions contained within a module A, the set of actions which will fire F, and the order in which they do so. The first step in this process is to determine which actions are "available." We may define a subset of actions, $\mathcal{A} \subseteq A$, or those actions which are available to fire. $\mathcal{A}$ includes all rules at all times, but methods are only included if invoked by a supermodule during the clock cycle under examination. In BSV, each action can have a guard expression specified. In the absence of explicit specification, guards default to being `true`. Let us define a predicate $\mathcal{G}$, such that $\mathcal{G}(a)$ if and only if the guard expression defined for $a \in A$ will evaluate to `true` in the given clock cycle.

We may then refine our set of actions using the above predicates to calculate $\mathcal{F} \subseteq \mathcal{A}$, the set of actions which may "fire" during a clock cycle.

$$\forall a \in A \bullet\ \mathcal{A}(a) \wedge \mathcal{G}(a) \mid \mathcal{F}(a)$$

It is important to note that $\mathcal{F}$ does not indicate the set of actions which *will* fire, only those that *may* fire. Actions must still succeed in arbitration in order to fire. The BSV action arbitration mechanism is designed to prevent race conditions while also attempting to maximize the number of actions executed in a particular clock cycle (Bluespec Inc., 2012a). This final step is non-trivial, as we shall see in §4.2, and throughout the bulk of this chapter.

Untimed vs Timed Semantics

In order to discuss action arbitration intelligently, we must first examine both semantics for action interpretation presented in the Bluespec literature: the untimed and timed semantics. These semantics are not adversarial, though they may at first seem to be. In fact, both are applicable simultaneously, as will be described below.

The untimed semantics presented in the Bluespec training material (Bluespec Inc., 2012b) describes the process by which one action is selected to fire, and what happens when it does fire. We will refer to the resulting operation as a "step". Since there is no direct relationship between untimed steps and hardware clock cycles, it is impossible to accurately model the behaviour of hardware generated from Bluespec descriptions solely on the basis of the untimed semantics. All formal models of Bluespec discussed in §1.4.1 apply themselves to the untimed semantics, without making this crucial leap to the timed semantics.

The single-step semantics of BSV may be summarized briefly as follows. An action in BSV is composed of either state element updates or invocations of submodule methods. State element updates are composed of a state element selection, combined with an update expression. In implementation, update expressions are interpreted as combinational circuitry, deriving inputs from the module's back of registers, so we derive two interesting properties.

- All update calculations occur simultaneously in hardware.
- Values read from registers remain constant for the duration of a clock cycle.

Note that constancy is retained for an entire clock cycle, not a single step. State holding elements update only once per clock cycle, regardless as to how many rules fire. Thus, if more than one action requests write access to a register in a clock cycle, these requests must undergo arbitration as here described. In addition, BSV actions fire either completely, or not at all.

Once $\mathcal{F}$, the set of actions which may fire, has been determined, these actions must undergo arbitration to resolve any potential conflicts. This is achieved by consulting rules of action precedence. Higher precedence actions block the execution of lower precedence actions in cases of memory conflict.

Let us define a relation, $\mathcal{S} \subseteq A \times R$, relating actions to state elements. For $a \in A$ and $r \in R$, $(a, r) \in \mathcal{S}$ only if the action a makes a write request to state element r. Read requests do not need to be included here, since they do not cause conflicts.

We can therefore define conflicts as a set of unordered pairs $\mathcal{C} = (a_0, a_1)$, where $a_0, a_1 \in A$ and $\mathcal{S}(a_0) \cap \mathcal{S}(a_1) \neq \emptyset$. That is, if two actions write data into at least one common memory object (register, vector, FIFO, wire etc.), those actions are said to be in *conflict*. It should be noted that the BAPIP translator requires all conflicting actions to be “resolved” by rules of precedence.

Precedence is established either implicitly via access rules for certain state elements, or explicitly by user-defined pragmas. In either case, we may define precedence as an ordered pair $\mathcal{P} = (a_0, a_1)$, where $a_0, a_1 \in A$. Defining $\mathcal{P}_E$ as those pairs belonging to explicit precedence, and $\mathcal{P}_I$ as the same belonging to implicit precedence (both discussed below), we may say $\mathcal{P} = \mathcal{P}_E \cup \mathcal{P}_I$.

The rule of explicit precedence is that, if an action name is included in the list of actions specified inside a `descending_urgency` pragma, each action in the list is of higher precedence than every action subsequent to it in the list. For example, if the descending urgency list is (A, B, C), where A, B and C are actions then $\{(A, B), (A, C), (B, C)\} \subseteq \mathcal{P}_E$.

Implicit precedence derives either from the use of particular hardware elements, or by the use of methods. All methods take precedence over all rules. So, if $M \subseteq A$ is the set of all methods in a module, and $U \subseteq A$ is the set of all rules in a module, then:

$$\forall m \in M \bullet \ \forall u \in U \bullet \ (m, u) \in \mathcal{P}_I$$

The use of certain specialized hardware elements has side effects on action scheduling (though not on state), most particularly wires. For wires with default values (`DWire` in BSV), rules writing to wires are given precedence over those reading them. So, if $W \subseteq R$ is the set of wires defined in a BSV module, $\mathcal{W}_R \subseteq A \times W$ is a relation from actions to the wires that action reads from. Similarly, $\mathcal{W}_W \subseteq A \times W$ is a relation between actions and the wires that action writes to. Note that conflicts also occur between wires.

$$\forall a_0, a_1 \in A \bullet \ \forall w \in W \bullet \ \{(a_0, w), (a_1, w)\} \subseteq \mathcal{W}_W \mid (a_0, a_1) \in \mathcal{C}$$

Inclusion in $\mathcal{P}_I$ is determined either by two actions reading and writing to the same wire.

$$\forall a_0, a_1 \in A \bullet \ \forall w \in W \bullet \ (a_0, w) \in \mathcal{W}_R \wedge (a_1, w) \in \mathcal{W}_W \mid (a_0, a_1) \in \mathcal{P}_I$$

Wires are discussed more fully in §4.1.3. This analysis proceeds similarly for FIFO enqueue and dequeue operations.

In effect, $\mathcal{P}$ describes a partial order over A. While actions are executed simultaneously in hardware, this partial order forms the basis for a sequential algorithm which determines the contents of F (those actions selected to fire). This ordering is identical with the concept of action scheduling.

Scheduling begins with the assumption that $\mathcal{F} = F$, and removes elements from F as required by the conflict resolution mechanism. At this stage, the translation process will have failed unless $\mathcal{C} \subseteq \mathcal{P}$, and will fail during scheduling if $\mathcal{P}$ contains cycles (indicating circular precedence). Scheduling proceeds by examining the contents of $\mathcal{F}$ one at a time. The partial order given by $\mathcal{P}$ is examined for maximal elements, and if $\mathcal{P}$ contains no cycles, there will always be at least one.

If multiple maximal actions exist, one is selected arbitrarily. In order for $a_0, a_1 \in A$ to both be maximal, $\{(a_0, a_1), (a_1, a_0)\} \cap \mathcal{P} = \emptyset$. Since $\mathcal{C} \subseteq \mathcal{P}$, we may deduce that these actions can not be in conflict. In hardware, if two actions do not conflict, and no precedence relates them, they are completely independent, simultaneous circuits. As such, it makes no difference which is selected to be scheduled first, the outcome will be the same.

Using the above selection mechanism, elements of $\mathcal{F}$ are examined one-at-a-time, and sorted into two sets, $F \subseteq \mathcal{F}$, the set of actions that will in fact fire, and $F' \subseteq \mathcal{F}$, the set of actions that will not fire. All actions must be sorted in to these two subsets, so that, by the end, $F \cup F' = \mathcal{F}$

If $\mathcal{G}(m)$, that is, the guard expression of m evaluates as true, then $m \in F$ and $\mathcal{C}(m) \subseteq F'$. If $\neg\mathcal{G}(m)$, then $m \in F'$ only, and no decision is made as to how the actions in $\mathcal{C}(m)$ are sorted. It should be noted that, once an action has been sorted into F', it is also considered to have been examined, from the perspective of examining each maximal element of $\mathcal{F}$, in order of maximality.

Repeated invocation of this selection mechanism composes some number of untimed steps into a single clock cycle, and is complete when there are no actions left to consider.

4.1.3 Wire and FIFO Semantics

Wires are an indispensable element of hardware design, used when a calculated result must be passed on in less than a clock cycle. Wires are exemplary of a small number of BSV constructs which violate the assumption of state output consistency within clock cycles. This property requires a lack of communication between the results of one action and another. It is only by maintaining this state consistency and other properties that action scheduling may be dealt with in so abstract a manner.

Syntactically, wire read and write operations are precisely the same as register read and write operations. The only differences exist semantically, and during instantiation. Wires make use of a range of constructors, but for this work only `DWire`, or wires with default values, are considered. This is because `DWire` is used exclusively in the real-world examples presented in §6.

The following code snippet gives syntax for wire declaration (line 2), reading (line 4), and writing (line 6). The primary purpose of this example is to demonstrate that, aside from a different initializer being used during declaration, and different type information, wires are used in precisely the same manner as registers.

BSV

```
DWire#(Bool) w1 <- mkDWire(False); // Wire Declaration
x <= w1;                           // Wire Read Operation
w1 <= True;                        // Wire Write Operation
```

END BSV

Wires are specifically a mechanism for communication of data between the actions of a module during a single clock cycle. Wires express data lines in a hardware design with no intervening state holding elements. Thus, they are purely combinational circuits. In Verilog, wires are handled using `assign` statements, and exist outside the `always` blocks in which registers are typically written to. In BSV, wires and registers may be used together within the same action; the semantic implications of using purely combinational circuits is enforced at the level of the scheduler.

Specifically, BSV will try to ensure that wires have values written to them before they are read from. If this is not possible, a `DWire` will substitute a specified default value. In this manner, wire scheduling introduces an ordering of precedence over the actions in a module without that ordering being backed up by rule pre-emption. That is, `DWire`s will never block a rule from executing, but they do change the order in which rules execute. Specifically, the scheduler puts rules with `DWire` write operations before rules with `DWire` read operations in the schedule. This will have important knock-on effects to overall scheduling, and can even be used for conflict resolution, so it is important to consider them when simulating BSV's scheduling mechanism.

It should be noted that FIFOs are also in this category. A FIFO, or a First In First Out buffer, is another essential hardware construct. It is a buffer, composed of registers, which may have elements queued and dequeued. The BSV scheduler will try to ensure that elements are queued in the buffer before being dequeued or cleared, and uses the same precedence ordering mechanism to do so as applies to wires. The various FIFO operations are discussed in more detail in §4.2.3.

Both wires and FIFOs violate one of the core abstraction principles of BSV, the one-rule-at-a-time semantics previously discussed. Insofar as previous work has sought only to model these simpler semantics formally, wires and FIFOs have been excluded from consideration by such works. As such, our consideration of them in our semantic translation constitutes an original contribution of this research. In our translation, we model these intermediately determined values using let bindings.

4.2 An Arbitration Algorithm

One of the primary algorithmic problems solved by the translation algorithm, and a primary contribution of this work, is transition from the guarded-action semantic of BSV to the state based Kripke semantics adapted for PVS. The key is accurately reproducing the manner in which the BSV scheduler decomposes scheduling into iterations of untimed steps.

At a very high level, the following steps are performed in the following order to translate from action to state based semantics.

1. Implicit and explicit scheduling constraints are tabulated for each action.
2. The actions are used to compose a binary decision tree, which describes all possible state transitions, for all state elements.
3. The decision tree is traversed once for each state element (i.e., register, wire, or FIFO), and a specified decision tree is generated which contains only the relevant state transformations.

A number of refinements and optimizations have been necessary, which are discussed in §4.3.

4.2.1 A Running Example

The information to follow is the most abstract and difficult part of this work, and this was reflected in amount of time it took to develop. In order to demonstrate and contextualize the following description, a running example will now be introduced. We expand our traffic light example from §2.1 to include some new features to control a four-way intersection:

- A North-South road and an East-West road.
- Two sets of car lights corresponding to the above roads.
- Pedestrian crossing signals, in addition to the car signals.
- Pedestrian crossing request signals for both North-South and East-West.
- A system reset signal.

An overview of the system, along with light timings, is given in Figure 4.2. Both car signals are controlled by the same state machine in Figure 2.1, though the East-West direction has had a timing offset. Pedestrian signals are easily calculated from the state of the car signals of the same direction. In our implementation of `TrafficSignals`, the pedestrian signals are not even given their own state elements, but are determined directly from the state of the car signal registers.

Our new `TrafficSignals` package is given below. Several changes have been made from the `TrafficLight` package presented in §2.1. Whereas `TrafficLight` relied on an external counter to time light transitions, `TrafficSignals` builds this functionality in. Below, literals of the form `N‘dXXX` are sized decimal integers, where `N` indicates the bitwidth of the literal. Annotations of this kind make proofs easier, but are not necessary to BSV semantics.

BSV

```
package TrafficSignals;

  interface TrafficSignals ;
    method Action reset ();
    method Action pedestrian_request_NS();
    method Action pedestrian_request_EW();
    method Bit#(2) getlamp_NS();
    method Bit#(2) getlamp_EW();
    method Bool getPedestrianLamp_NS();
    method Bool getPedestrianLamp_EW();
```

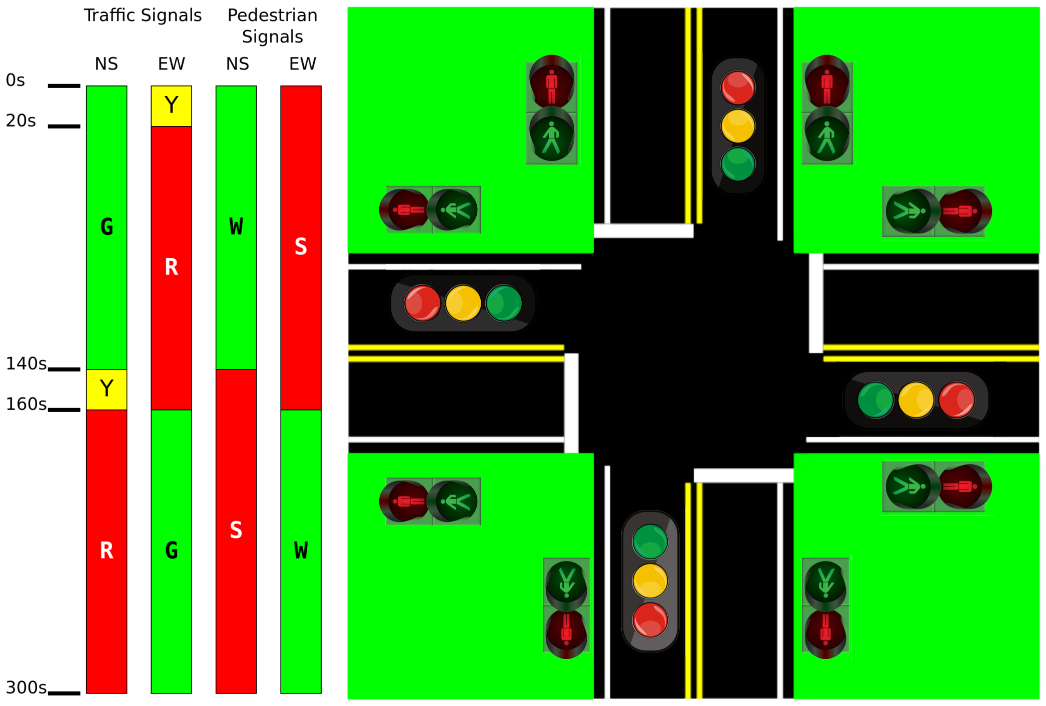


Figure 4.2: Intersection Signal System Overview

```
endinterface

(*descending_urgency = "reset,pedestrian_request_NS,
    pedestrian_request_EW"*)
module mkTrafficSignals (TrafficSignals);
  Reg#(Bit#(2)) carLamps_NS <− mkReg(2);
  Reg#(Bit#(2)) carLamps_EW <− mkReg(2);
  Reg#(Bit#(9)) t <− mkReg(0);

  rule tick;
    if (t < 300)
      t <= t + 1;
    else
      t <= 0;
  endrule

  rule goYellow_NS((carLamps_NS==2'd0)&&(t==9'd140));
```

```
    carLamps_NS <= 1;
  endrule

  rule goRed_NS((carLamps_NS==2'd1)&&(t==9'd160));
    carLamps_NS <= 2;
  endrule

  rule goGreen_NS((carLamps_NS==2'd2)&&(t==9'd0));
    carLamps_NS <= 0;
  endrule

  rule goYellow_EW((carLamps_EW==2'd0)&&(t==9'd0));
    carLamps_EW <= 1;
  endrule

  rule goRed_EW((carLamps_EW==2'd1)&&(t==9'd20));
    carLamps_EW <= 2;
  endrule

  rule goGreen_EW((carLamps_EW==2'd2)&&(t==9'd160));
    carLamps_EW <= 0;
  endrule

  method Action reset ();
    carLamps_NS <= 2;
    carLamps_EW <= 2;
    t <= 0;
  endmethod

  method Action pedestrian_request_NS();
    if (carLamps_EW == 0 && t < 9'd280)
      t <= 280;
  endmethod

  method Action pedestrian_request_EW();
    if (carLamps_NS == 0 && t < 9'd120)
      t <= 120;
  endmethod

  method Bit#(2) getlamp_NS();
```

```
      return carLamps_NS;
    endmethod

    method Bit#(2) getlamp_EW();
      return carLamps_EW;
    endmethod

    method Bool getPedestrianLamp_NS();
      return carLamps_NS == 0;
    endmethod

    method Bool getPedestrianLamp_EW();
      return carLamps_EW == 0;
    endmethod

  endmodule : mkTrafficSignals

endpackage : TrafficSignals
```

END BSV

As any pedestrian can attest, whether the pedestrian crossing request signals work depends on when the button is pressed. A request only works if received during a red light, with more than 20 seconds remaining until the light goes yellow. The effect of the request is to "move up" the timer, so that the time remaining during the red light is reduced to twenty seconds. This delay allows car traffic some chance of clearing before the light transition, and instills doubt in the mind of the pedestrian as to whether the button even works, which is a universal feature of pedestrian crossing request buttons in the author's home city. At all other times the signal has no effect.

4.2.2 The Determination of Conflicts

Action Scheduling in BSV requires the explication of two implicit properties of actions: confliction and precedence. From a practical standpoint, each action carries a (possibly empty) list of the other actions it conflicts with during scheduling. The property of confliction, explored in detail in §4.1.2, states that two actions are in conflict if and only if a) the set of state elements they write to is non-disjoint AND b) the Boolean guard expressions are also non-disjoint. In other words, actions do not conflict if they do not compete for writing privileges on at least one state element, and if they do, they only conflict if their guard expressions may both be True at the same time.

The first condition may be determined by extracting the set of state elements written to via analysis of the action's statements.

The second condition is less trivial. The disjointness of expressions is not a property which can be determined via purely syntactic analysis; but the problem is amenable to SMT (Satisfiability Modulo Theory) analysis (Dutertre and De Moura, 2006). While Boolean Satisfiability would suffice for expressions containing only Boolean values, guard expressions may contain any valid BSV type, such as integers and bit vectors. As such, SMT solving is necessary. The disjointness property is easily re-conceptualized as conjunctive satisfiability. That is, if we take the Boolean conjunction of the two expressions, is there some set of concrete variable values which yield true, when substituted into the conjunction of the guard expressions. Without solving the disjunction problem with respect to rule guards, no false conflicts can be eliminated, and classes of hardware descriptions that would in fact work would be excluded from modelability.

To this end, we use SBV, a Haskell library for symbolic SMT solving, to check the satisfiability of these guard constraints (Erkök, 2019). SMT-based verification in Haskell (SBV) is a Haskell-based front-end to a number of popular SMT solvers, including Z3 (De Moura and Bjørner, 2008), Yices (Dutertre and De Moura, 2006), MathSat (Cimatti et al., 2013), Boolector (Brummayer and Biere, 2009), and CVC4 (Barrett et al., 2011). Yices was adopted as our back-end SMT solver. Not only did it work satisfactorily and consistently, but, like PVS, is developed and maintained by SRI International. This necessitated the construction of an analysis engine which converts Boolean expressions represented in our BSV abstract syntax (§3) to the expression system of SBV, as well as the extraction of symbolic variables, and determination of their possible ranges based on type declarations. Fortunately, this latter information is well constrained, due to both the strong static typing of BSV and the inherent nature of hardware description.

Practically, the algorithm first identifies all potentially conflicting actions via state disjointness analysis. SMT solving is invoked to adjudicate whether a conflict is truly present. If the guards are not satisfiable in conjunction, the conflict is a false one, and the conflict is removed from the tabulated conflicts of the relevant actions. The SMT results have no further use after this point, and are not kept. If the conflict is authentic, the schedule checks that an order of precedence is established. If not, the BSV design is declared to have unresolved conflicts, and the translation process is terminated. A detailed explanation of why we treat this as a fatal error is given in §4.2.3.

Running Example

Our `TrafficSignals` hardware description contains a number of conflicts, as displayed in Figure 4.3. Based solely on an analysis of which actions write to which registers, `TrafficSignals` has three potential conflict groups, corresponding to the three registers instantiated by the module.

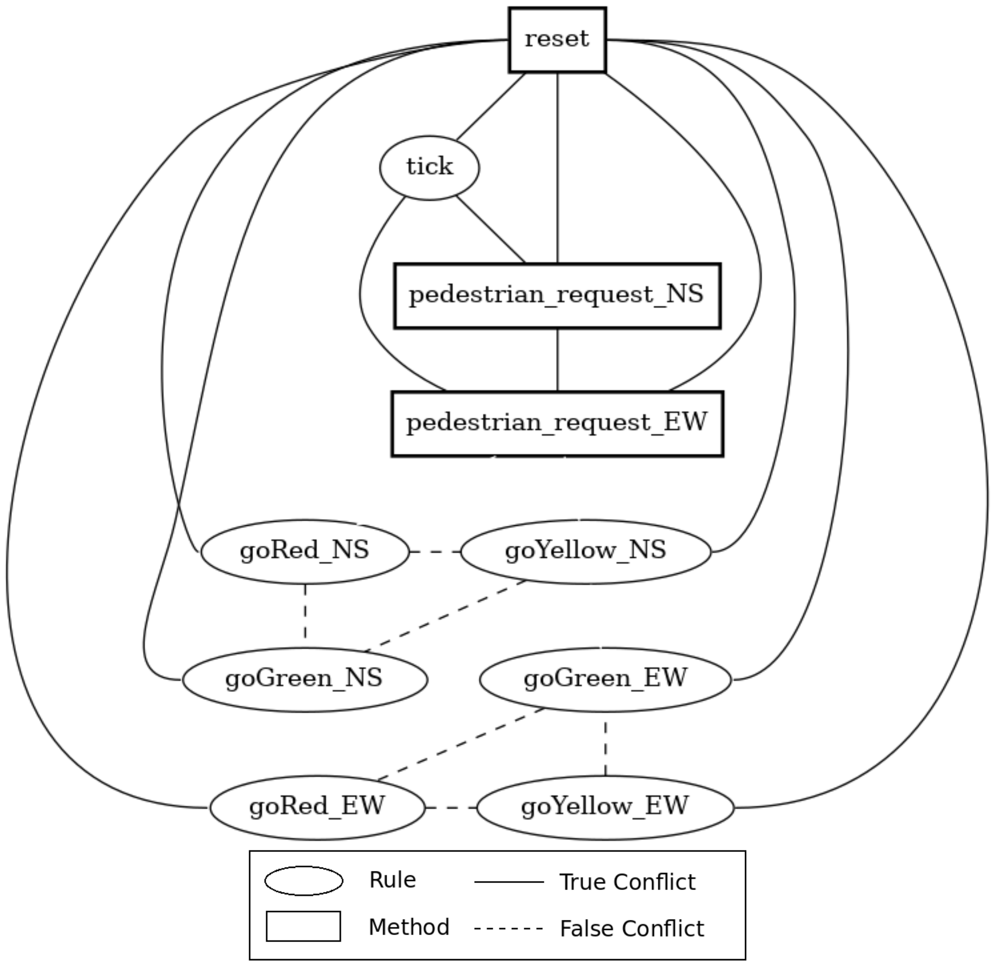


Figure 4.3: Intersection Example Conflict Graph

Some of these conflicts are not truly conflicts. Upon observation of their guard expressions, the rules controlling lamp state (goRed, goYellow, etc.) cannot be executed simultaneously. This is deduced by Yices and SBV, by testing to see if any particular combination of registers allows any two rules in the same conflict group to have guard expressions evaluating to True simultaneously. In this case, the lamp state alone is sufficient to demonstrate mutual exclusivity. For an example like this which considers only equality comparison, the argument could be made that SMT solving is overkill. The advantage of SMT solving is that we can deduce the satisfiability of expressions containing not only simple equality comparisons, but any arithmetic or logical operator supported by BSV.

Examination of the `reset` method tells a different story. By resetting each register, the `reset` method places itself in conflict with every other action. This makes sense of course, if a reset signal is received, the controller should do nothing except service the reset operation. However, in order to give `reset` this privileged position, we need to explore both the implicit and explicit mechanisms of precedence.

4.2.3 Conflict Resolution using Rules of Precedence

While conflict seems an inherently undesirable state, it is not necessarily the case that every conflict needs to be worried about. The secondary layer of action scheduling interactions is that of precedence. Precedence resolves conflict in a very straightforward manner. If two actions are in conflict, the action with the highest precedence is given the opportunity to execute first, preempting the other. This does not necessarily mean that the higher precedence action *will* execute, but it is given the opportunity to. In other words, conflict plus precedence plus execution equals preemption.

From a practical standpoint, it is only necessary for each action to know which actions may preempt it directly, and from this a partial order may be constructed over actions. Interpretation of this poset will be discussed in §4.2.4. The rules of precedence are as follows:

1. Methods take precedence over rules.

2. Precedence may be assigned manually using pragmas.

3. Actions in which DWires are written to take precedence over actions in which the same DWires are read from.

4. Actions containing methods operating on FIFOs have precedence implications (as shown in Fig 4.1).

In general, we refer to precedence rules declared by the user using pragmas as being *explicit*, since the rule of precedence is the primary result of user action. In all other cases, rules of precedence are secondary consequences of BSV semantics, and so are referred to as *implicit*.

Semantically, methods are identical to rules, except that they must be invoked by a supermodule in order to be eligible to fire in any particular clock cycle. When a method is invoked by a supermodule, it is given priority over all rules in that module. In our running example, this gives precedence to `reset`, `pedestrian_request_NS` and `pedestrian_request_EW` over `tick`, resolving three out of the six conflicts not resolved during mutual exclusivity checking.

It is useful at this point to elaborate method semantics somewhat. It is not possible for a method to be invoked more than once by different modules because each module, when instantiated, has only one supermodule. The modules in a BSV design form a strict hierarchy. It is possible for multiple actions in a single module which invoke a method to be queued for execution, but the write conflict resolution mechanism kicks in, so that both actions are prevented from executing in the same clock cycle. A method can therefore only be invoked once per clock cycle.

Methods are not guaranteed to be free of conflict, since it is possible to construct them such that multiple methods write to the same state element. While it is generally bad practice to create conflicting methods, this is also true of rules, and invoking multiple methods in one clock cycle is perfectly legal in BSV, so conflicts are not resolved by only being able to call one at a time. Such cases are handled the same as rules, using the same implicit or explicit rules of precedence, and BAPIP will fail to translate unresolved method conflicts for the same reason as it fails on unresolved rule conflicts. The only distinction between methods and rules with respect to scheduling is that a method must be queued for execution by the supermodule, and methods are implicitly higher priority than rules. After these conditions are established, the normal mechanism takes over.

A DWire is a wire with a default value. Wires are impermanent data containers which, in terms of hardware, can be thought of literally as the connection of the result of some combinational circuit to some other part of the design, with identifier labelling for ease of use. A default wire has a default value, if it is accessed before it has been written to, but for every declared wire, the action in which that wire is written to has implicit precedence over any action reading it. In effect, each declared DWire carries within an implicit statement of precedence, dependent on the manner of its usage, which may be used to adjudicate conflicts.

Scheduling Annotations mkFIFO, mkSizedFIFO, mkFIFO1				
	enq	first	deq	clear
enq	C	CF	CF	SB
first	CF	CF	SB	SB
deq	CF	SA	C	SB
clear	SA	SA	SA	SBR

C	Conflict
CF	Conflict and Preemption Free
SB	Scheduled Before
SA	Scheduled After
SBR	Scheduled Before (Restricted[1])

Tab. 4.1: Scheduling Implications for FIFO Method Invocations (Bluespec Inc., 2012a)

In Table 4.1, we have a delineation of all of the scheduling implications introduced by the use of the four basic FIFO methods (`enq`, `deq`, `clear`, and `first`). Table 4.1 is meant to be read as "row method has X relation to the column method".

In this manner we may see that, for example, use of a `clear` operation implies that the clearing action is of lower precedence than all other actions invoking any other method on the same FIFO. Note also that the state altering operations `enq` and `deq` do not conflict with each other. As with wires, there are a number of types of FIFO available in BSV, and many of these are distinct not from functional or descriptive differences, but due to different scheduling implications.

In the failure of all implicit forms of precedence assignment, the user is given an explicit precedence mechanism, in the form of the `descending urgency` pragma. Invoked anywhere within or immediately adjacent a module, this pragma takes a list of rule names, and assigns them descending precedence. This is the most efficient manner of resolving unresolved conflicts which the translation algorithm fails on, though the presence of unresolved conflicts may mean the design should be re-examined.

In our example, the conflict between our three methods can not be solved by implicit precedence mechanisms. Thus, a descending urgency pragma is used to break the tie. It is reasonable that a reset operation should have the highest level of priority, and it is given such. The ordering between the other two is selected arbitrarily. This explicit precedence assignment resolves the remaining three conflicts that went unresolved during mutual exclusivity checking.

Tabulating Actions

In order to transform BSV's action centric semantics into state centric semantics, it is necessary to illustrate the manner in which actions are tabulated. This will give some insight into the details that the translation algorithm must gather before schedule generation.

For each action in the hierarchy of modules, one of the following record structures in Haskell are put together. The reason this structure is referred to as a rule schedule and not an action schedule, is that the translator contains a preprocessing step which converts all non-rule actions to rules. The most work invoked in this transformation is keeping track of the method arguments for later use by the state transition predicates. By the time we get to this stage, every action is encoded as a rule in BAPIP. The following Haskell code is a record data type containing all information BAPIP collects about an action.

HASKELL

```haskell
data RuleSchedule = RuleSchedule{
    rName               :: ActionPath
      -- Action's Identifier
  , rGuard              :: Expression
      -- Guard Expression
  , rStatements         :: [Statement]
      -- Contained Statements
  , implicitConditions  :: [Expression]
      -- Guards of Called Methods
  , writesTo            :: [ID_Path]
      -- Registers Written To
  , dWireWrites          :: [ID_Path]
      -- DWires Written To
  , dWireReads           :: [ID_Path]
      -- DWires Read From
  , fifoEnqs             :: [ID_Path]
      -- FIFO enq() Calls
  , fifoDeqs             :: [ID_Path]
      -- FIFO deq() Calls
  , fifoClears           :: [ID_Path]
      -- FIFO clear() Calls
  , fifoFirsts           :: [ID_Path]
      -- FIFO first() Calls
  , actionMethodsCalled :: [Expression]
      -- Invoked Action Methods
  , conflictsWith        :: [ActionPath]
      -- Actions I conflict with
  , noConflictsWith      :: [ActionPath]
      -- Actions I don't conflict with
  , preempts             :: [ActionPath]
      -- Actions I preempt
  , isPreemptedBy        :: [ActionPath]
      -- Actions I don't preempt
  , executesAfter        :: [ActionPath]
      -- Actions I excute after
  , executesBefore       :: [ActionPath]
      -- Actions I execute before
  } deriving (Eq)
```

END HASKELL

Why Unresolved Conflicts are a Sufficiently Bad Problem

It is important to note that the algorithm will fail in its translation if any two actions are determined to have an unresolved conflict. This may seem like an over-reaction, but it is the only way to guarantee the correctness of the translation algorithm, for reasons that will now be illustrated. In the BSV documentation (Bluespec Inc., 2012a), it is explicitly stated that conflicts are resolved by some "arbitrary but deterministic" mechanism, the details of which are not published. While the idea of this arbitration being deterministic is a tantalizing clue as to the operation of the scheduler, the deliberate lack of information about the scheduling algorithm means that it is impossible for us to determine which action would win an ambiguous conflict, regardless of how repeatable the result may be. Previous authors addressing this problem seem to have interpreted this as "any action may fire," and excused themselves from the difficulty of reproducing the clock cycle composition semantics (Richards and Lester, 2011). In order to usefully demonstrate the correctness of a class of theorems larger than invariants over single untimed steps, it is necessary to precisely emulate full clock cycle semantics. Theorems involving timing requirements are one such example. Therefore, it is necessary to constrain the design to completely unambiguous schedules. The translation algorithm will therefore fail if unresolved conflicts are detected.

In our running example, we observe that `pedestrian_request_NS` and `reset` are in conflict with each other. The state of register `t` is dependent on the execution order of these two actions (see Figure 4.3 above). Since we can say with confidence that their guard expressions are not disjoint, both being the truth literal, this is a "true conflict". It is impossible to determine with consistency which of these two rules would be selected for execution on any given clock cycle, due to the "arbitrary but deterministic" nature of conflict resolution. Therefore, we have two possibilities: either `pedestrian_request_NS` goes first, or `reset` does. This can be deterministically calculated from the evaluation of action guards and the confliction and precedence graphs If a formal determination of the outcome of the schedule is to be made, we *must* know which is considered for execution first. Otherwise, the number of possible valid schedules grows with each such ambiguous conflict. While it would be theoretically possible to keep track of all such possible schedules, this would have a disastrous effect on proof execution times, and may not even be desirable.

The additional argument can be made that, in practice, the presence of unresolved and underdetermined action scheduling constitutes bad design on the part of the hardware designer. If one is being sufficiently conscious of modularization and information hiding principles, one would take care that state elements are being written to in as few places as possible. This design principle is mostly adhered to in the RapidIO subsystem discussed in §2.4, minimizing conflicts.

4.2.4 Generating a Universal Schedule

All combinations of action firings with unique effects on the state, as well as state transformations, require a data structure. Examination of this data structure is also informative with respect to understanding the design of the algorithm.

The universal schedule is a binary tree which describes all possible action execution sequences, and encapsulates all information necessary for the generation of scheduling trees for each individual state element. In other words, all guards and statements are collected into the universal tree for later reference.

Each node in this tree is an action, with a branch for the consequences of its execution, and a branch for the consequences of its non-execution. The root of the tree must be maximal with respect to all the precedence orderings discussed in §4.2.3.

The universal scheduling tree is created via the following recursive graph algorithm, which uses as input the conflict and precedence posets as described above. Given a set of actions and their scheduling implications, one action is selected as "next to fire." This action must be a maximal element of the partial order imposed by implicit and explicit precedence. Mechanically, a maximal action is an action whose list of actions preceding it is empty. If there are multiple maximal actions any one of them may be selected arbitrarily. For actions which do not conflict, and for which no order of precedence is established, the order in which those actions are selected for execution has no effect on the state once all such actions have been executed. In hardware, such actions are implemented as parallel circuits which perform calculations simultaneously, but the degree to which assuming consecutive instantaneous execution simplifies the semantics underlying the translator can not be overstated. This reasoning is based on the atomic, parallel nature of BSV semantics. Each statement is a combinational circuit, and all values stored in registers are only updated at the end of a clock cycle. The translation algorithm trivially selects the first action in the list of actions, which corresponds to the action which appears first in the source BSV file. Semantically, any would do. The guard expression and action statements are recorded in the tree. Two modified lists of action schedules are produced, which will be used to generate the true and false branches of this node. For the false branch, the list of actions is reduced by the action selected, and all references to the maximum are deleted from all other actions' lists of preceding and conflicting actions. For the true branch, the further step is taken of removing from the list all actions which were in conflict with the maximum, since the execution of an action blocks the execution of all actions the chosen action conflicts with. Thus, the true and false action lists are passed to two instances of the tree generation algorithm recursively. The algorithm terminates when no actions are remaining to be scheduled.

The purpose of the universal schedule is to serve as an intermediary representation between the action oriented rule tabulation read more or less directly from the BSV source files, and the state-centric trees for each state element that are necessary to generate branching if expressions for each state element (Registers, FIFOs, and DWires). These if expressions are eventually collected into record update predicates, and form the transition predicates of our Kripke model.

A Data Structure for Schedules

Up to this point, it has been sufficient to store our actions, plus associated scheduling information, as a tabulation. Each action is parsed into a record, and these records are analyzed to construct lists of other actions which conflict or pre-empt. The conversion from this tabulation of actions, conflicts, and precedences into a hierarchical organization of precedence constitutes the actual algorithmic transformation from an action oriented semantic to a state-based semantic.

The generation of a universal schedule of course necessitates a universal scheduling algorithm which produces it. This algorithm is discussed in detail in §4.2.4. Before we discuss it, however, it is important to clearly describe the form of the generated data structure. A universal schedule data type is represented in Haskell as follows, where `Statement` is defined in §3.1.5.

HASKELL

```haskell
data TotalTree
 = TotalStem Guard [Statement] TrueTree FalseTree
 | TotalLeaf Guard [Statement]
 deriving (Eq)

type TrueTree = TotalTree
type FalseTree = TotalTree
```

END HASKELL

In other words, a `TotalTree`, or universal schedule, may be either a stem node or a leaf node. Stem nodes contain a guard expression and a list of statements derived from one of the tabulated actions, as well as two more universal schedules. One assumes the guard expression was true, and describes the rest of the action decisions that follow from that assumption. The other assumes the guard expression did not evaluate true, and describes the rest of the action decisions contingent on that outcome. A leaf node, on the other hand, contains only the action information, with no subtrees defined. Leaf nodes will typically only occur when the universal scheduler is scheduling the final action. In the Haskell, the second and third line merely identify the `TrueTree` and `FalseTree` as type synonyms for `TotalTree`.

The Generation Algorithm

Running Example

The extraction of a partially ordered set of actions from a BSV file is a matter of relatively straightforward static analysis. A list of the register and wire interactions performed by each action is extracted, and from this information lists of rule conflicts and the (implicit) wire ordering may be extracted. The primary method of direct schedule manipulation in BSV is pragma declaration. The user may force the scheduler to deal with rules as if they will never be available to fire simultaneously using the `mutually_exclusive` pragma, or set explicit precedence orderings using `descending_urgency`. There are many pragmas available, a full listing is provided at (Bluespec Inc., 2012a).

Pragmas are an extremely powerful mechanism for schedule determination. They have the power to override not only implicit precedence calculations, but conflicts themselves. With pragmas such as `mutually_exclusive` and `conflict_free`, we can override every stage of the scheduling algorithm. The `mutually_exclusive` pragma asserts that two actions will never be simultaneously available for execution, even if they can be. The `conflict_free` pragma asserts that two actions do not conflict, even if they do. Overuse of pragmas can therefore result in the creation of race conditions, so caution is advised. Within the context of our running example, any conflict between actions could in theory be immediately resolved using a `conflict_free` pragma, but this may result in an unstable state, unless the fact the two actions are conflict free is provable by the programmer. In a sense, the user would need advanced knowledge of a design's operation in order to correctly make such assertions.

If any conflicts remain unresolved by any added pragmas, BAPIP will terminate, displaying an error requesting additional pragmas in the source BSV file. The example presented here contains no unresolved conflicts, but some could easily be created by removing the `descending_urgency` pragma.

For our example file `TrafficSignals.bsv`, Figure 4.4 is the resulting partial ordering for all those actions with true conflicts. In this diagram, unidirectional arrows point from high precedence to low precedence actions.

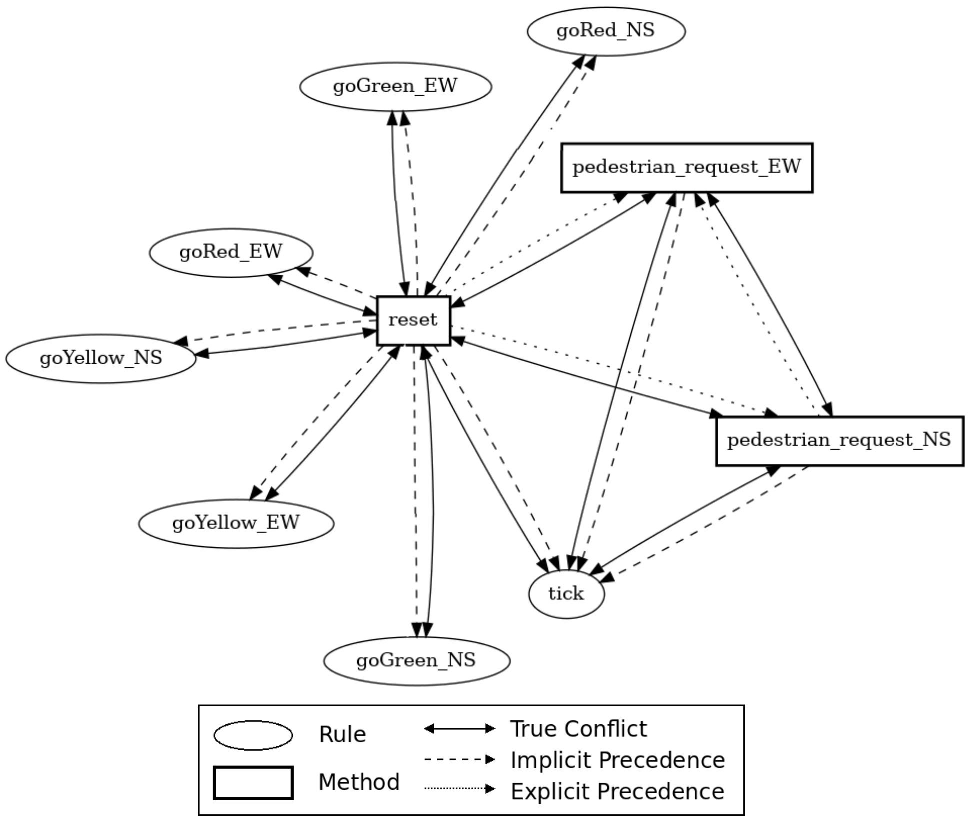


Figure 4.4: Intersection Example Partially Ordered Set

During scheduling, all three methods take precedence over all rules, but in the above graph, only those implicit precedence relationships which resolve rules have been depicted. This is to avoid cluttering the graph unnecessarily.

Following the greedy algorithm approach outlined in §4.2.4, we must now transform this action poset into a decision tree. A partial elaboration of the scheduling algorithm is given in Figure 4.5. Each node of this graph contains the state of the action poset remaining at that stage. The action selected by the scheduler at each stage is highlighted in grey.

Our top-level poset contains all ten actions. Given its maximal position in both implicit and explicit precedence, `reset` is the first rule to be selected for scheduling. Each node is a *decision* about an action firing, with a subtree both for the instance of it firing and of it failing to fire. Since `reset` conflicts with every other action, if executed, no actions will remain to be scheduled. If it does not execute, the node along the true branch receives a poset with `reset` removed.

The next action to be scheduled is `pedestrian_request_NS`. At this stage, the colour changing rules are not eligible for firing, because methods take precedence over rules. From here, the action firing will result in only the colour changing rules being left, since `pedestrian_request_NS` conflicts with all the others. At any point when the poset is free of conflicts, the remainder of the tree is omitted. These proceed via rudimentary one-at-a-time selection, which need not be illustrated. Further, the same series of selections would be repeated a number of times. After this, `pedestrian_request_EW` and `tick` are selected along the false paths via the same mechanism.

Reading only the highlighted actions in the above tree yields the universal scheduling tree.

In the next section, we demonstrate how this universal schedule is easily transformed into state element specific schedules, which have a one-to-one mapping with branching if-statement structures implementable in PVS.

4.2.5 State-Oriented Universal Schedule Interpretation

A Data Structure for State Specific Schedules

Once the universal schedule has been created, all that remains is the refactoring of this schedule into a decision tree for each specific state element. Our model of computation in PVS is the state-centric Kripke structure. The arrangement of state in any given clock cycle is transformed via the scheduling algorithm into the next clock cycle's state. Now that we have elaborated the universal schedule, we must use it to generate transition expressions for each individual state element, which necessitates the creation of state-specific scheduling trees.

As the universal schedule is traversed, all data pertaining to the particular

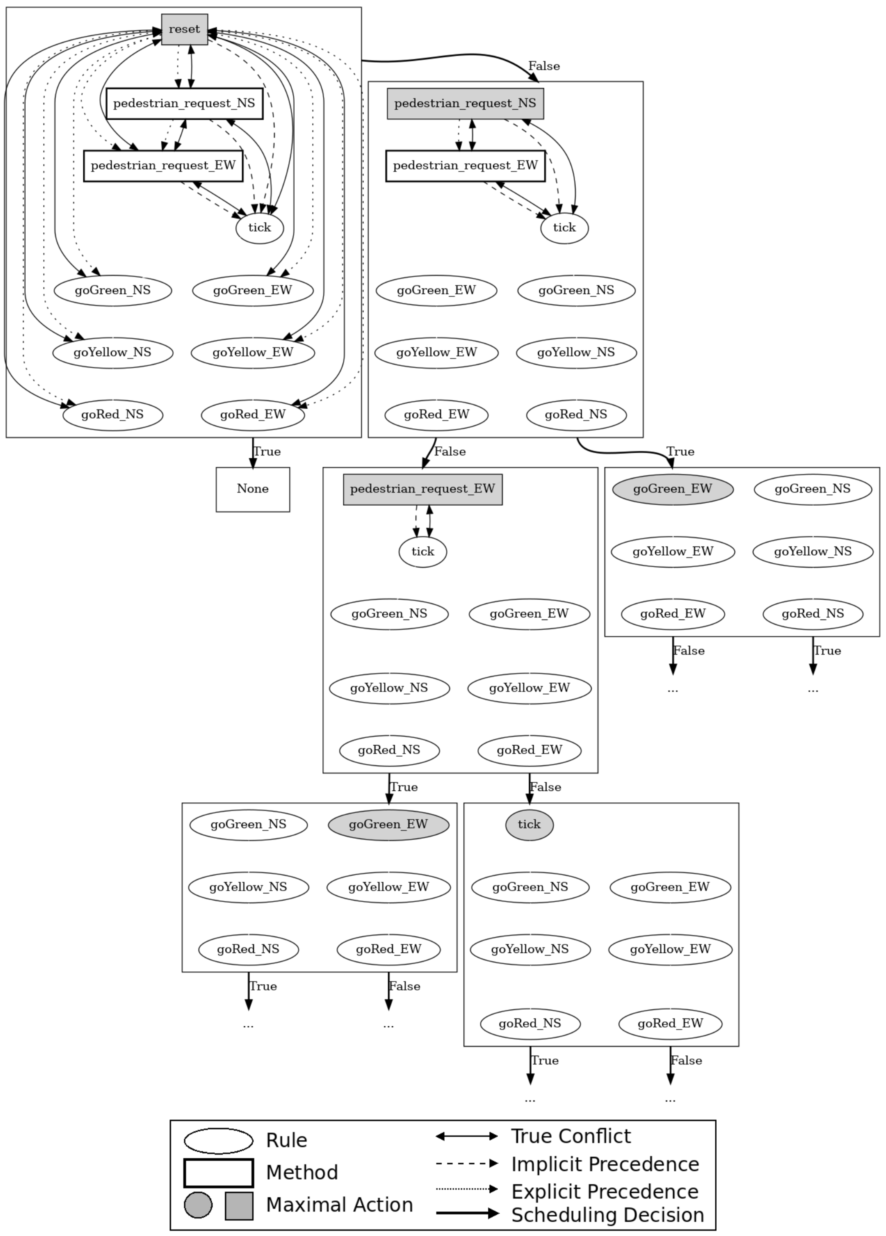


Figure 4.5: Intersection Example Universal Schedule

state element under examination will be collected into the following data structure.

HASKELL

```
data SpecificTree = SpecStem Guard
          (Either Expression SpecTrueTree) SpecFalseTree
  | SpecLeaf Guard TrueExpression FalseExpression
  | SpecEx Expression
 deriving (Eq)
type SpecTrueTree = SpecificTree
type SpecFalseTree = SpecificTree
type TrueExpression = Expression
type FalseExpression = Expression
```

END HASKELL

The `SpecificTree` is a tree specific to some particular state element. Like the universal tree, we recognize stem nodes and leaf nodes. `SpecStem` contains the action's guard expression and a false sub-tree. Rather than simply pointing to a true sub-tree, the specific tree recognizes that, if the current action's guard is true and that action writes to the relevant state element, there is no need to continue keeping track of the rest of that branch of the tree. Within the data structure, the true sub-tree may be replaced with an expression (the value to be written), using Haskell's `Either` datatype. Otherwise, `SpecLeaf` merely substitutes the expression to be written for a list of statements on both true and false guard results. The false result is always a status quo result, which in PVS is interpreted as a state element being written with its previous value. The `SpecEx` node is used in rare cases when an expression needs to be substituted, but has no associated guard.

Generating the State Specific Schedules

Recalling that each node in the universal scheduling tree corresponds to an action described in the underlying BSV description, to produce a scheduling tree for a specific state element requires only the exclusion of all write operations that do not write to the state element under consideration. This can be accomplished by a straightforward traversal of the universal scheduling tree, and a recording of the results in the above data structure.

These trees, of which one is generated for each state element, are subsequently simplified and transformed into branching if-statements during PVS generation.

Running Example

The state assignment statements contained in each action are kept track of within the generated universal scheduling tree. This produces a tree that must be converted from the action-centric semantics of BSV to the state-centric representation in PVS. Fortunately, a version specific to each state element can be generated by traversal.

Once state specific trees have been generated, they may be pretty-printed as if expressions and inserted into the generated transition predicates, which are encoded in record-update syntax. The following code listing is an excerpt of `Transitions.pvs`, generated from our example file. The full example may be found in Appendix C.2.4. The syntax below is that of a PVS record update. In this case, `pre` is a record holding the previous state in our finite state machine. The `with` operation in PVS takes a previous record and updates those fields for which updates are specified using assignment (in this case all three). The resulting record is the post state, which may be used by the theorem prover and checked for properties of interest.

PVS

```
%% \gls{pvs} excerpt – Transition with call to reset() %%
transition_val (index : nat, pre : TrafficSignals) : TrafficSignals =
    pre with
        [ t := 0
        , carLamps_EW := 2
        , carLamps_NS := 2
        ]

%% \gls{pvs} excerpt – Transition without call to reset() %%
transition_val (index : nat, pre : TrafficSignals) : TrafficSignals =
 pre with
  [ t := if ( pre`t < 300 )
      then ( pre`t + 1 )
      else 0
    endif
  , carLamps_EW:= if(((pre`carLamps_EW=2) AND (pre`t=160)))
      then 0
      else if (((pre`carLamps_EW=1) AND (pre`t=20)))
        then 2
        else if (((pre`carLamps_EW=0) AND (pre`t=0)))
          then 1
          else pre‘carLamps_EW
        endif
      endif
    endif
  , carLamps_NS := if (((pre‘carLamps_NS=2) AND (pre‘t=0)))
      then 0
      else if (((pre‘carLamps_NS=1) AND (pre‘t=160)))
        then 2
        else if (((pre‘carLamps_NS=0) AND (pre‘t=140)))
          then 1
          else pre‘carLamps_NS
        endif
      endif
    endif
  ]
```

END PVS

The first excerpt is the record update corresponding to a transition in which the reset method has been called. One may recall that the `reset` method had no guard expression in the original description, which defaults to a truth literal. Although the full tree would have originally been expressed for all three fields along the false branch of the various state specific scheduling trees, the presence of a truth literal at the top level allows the tree to be simplified down to a single statement in the case of each register. The full decision tree has a maximum depth equal to the number of actions in a module. In this case, ten actions would yield a tree with a possible maximum size of 1024 nodes, each of which would occupy a line of code. Without the simplifications, simply displaying the full if-tree would be a dubious prospect in so confined a document as a PhD thesis.

Our second excerpt is a transition in which no methods are called. As such, we can see how the algorithm has removed rules which, although necessarily existing in the universal schedule, contribute nothing to the state element at hand. What results is highly readable code, which has a direct correspondence to the original code.

4.2.6 Comparison to Richards and Lester Method

In order to demonstrate the superiority of this approach, let us discuss how the Richards and Lester (2011) approach would be applied to our running example. It should be noted before we begin that the Richards and Lester method is a manual method, so it is not possible to easily produce an exact translation of our example into the Richards and Lester method. However, it is instructive to include some small code example from their published work. The code is the same used to support their 2011 paper (Richards and Lester, 2011), and can be retrieved from the internet (Richards, 2011a).

The Richards and Lester method (hereafter referred to as the monadic method) does not attempt any simulation of the BSV action arbitration mechanism, but stops at modelling the individual actions as individual state transitions. The greatest strength of this method is that, using a monadic approach, these state transitions are syntactically very similar to the BSV actions they are based on. This is somewhat deceiving, however, as these monadic actions require a not inconsequential amount of supporting definitions, which the authors hide away in imported theories. These supporting definitions are not universally applicable, but must be updated as the monadic action implementations are added or modified, so the claim that the monadic method's actions are minimal syntactic manipulations of the original BSV methods is not well supported by the evidence. The following code snippet represents one rule from their "Peterson" example (Richards and Lester, 2011).

PVS

```
%%%%%%%% Original \gls{bsv} Code by Richards and Lester
    %%%%%%%%
% rule q_critical  (pcq == Critical && p_data.notFull);
%   p_data.enq (False);
%   pcq._write (Sleeping);
%   turn._write (True);
% endrule

  q_critical  : Rule
    = rule (pcq‘read = Critical  and  fifo ‘enq_cond,
%             ------------------------------------
              fifo ‘enq    (true)      >>
              pcq‘write    (Sleeping)  >>
              turn‘write   (true))
```

END PVS

As we can see, their claim of close syntactic correspondence is superficially true. Let us therefore examine the supporting definitions, taken from across three other files (Richards and Lester, 2011).

PVS

```
get_pcq  (p: Peterson): Reg  [PC]   = p`pcq
get_turn (p: Peterson): Reg  [bool] = p`turn
get_fifo (p: Peterson): FIFO1[bool] = p`fifo

update_pcq (p: Peterson, r: Reg[PC]    ): Peterson
  = p with [(pcq ) := r]
update_turn (p: Peterson, r: Reg[bool]  ): Peterson
  = p with [(turn) := r]
update_fifo (p: Peterson, r: FIFO1[bool]): Peterson
  = p with [( fifo ) := r]

pcqT  : Transformer [Reg[PC],      Peterson, T] =
  transform (get_pcq, update_pcq )
turnT : Transformer [Reg[bool],    Peterson, T] =
  transform (get_turn, update_turn)
fifoT  : Transformer [FIFO1[bool], Peterson, T] =
  transform (get_fifo , update_fifo)

pcq   : RegFunctions   [Peterson, PC]   =
  getRegFunctions   (pcqT [PC],   pcqT [Null])
fifo  : FIFO1Functions [Peterson, bool] =
  getFIFO1Functions (fifoT[bool], fifoT [bool],  fifoT [Null])
turn  : RegFunctions   [Peterson, bool] =
  getRegFunctions   (turnT[bool], turnT[Null])
```

END PVS

This does not include the state record definition, or five files of prelude material, wherein the Bluespec Monad itself is defined, as well as read and write operations.

In the case of our example, we would produce a total of ten monadic action embeddings, one for each rule, and one for the method in our original BSV description. Each one of these monadic actions would perform the work of precisely one action in transforming the state, if applied to the module's state record. Individually, these monadic actions are capable of proving some properties. In particular, this approach is useful if one needs to demonstrate an invariant in the system. One could reasonably simply compose a theory which ensures that no action or combination of actions modifies the state in such a way that the invariant is violated.

The way that Richards and Lester compose their monadic actions into a single transition predicate is through the *simple disjunction of all monadic actions*. The following code snippet, taken again from the Richards and Lester (2011) Peterson example is all that is given for a transition predicate.

PVS

```
transitions (pre, post) : bool =
     wake_p     (pre, post)
  or wake_q     (pre, post)
  or grant_p    (pre, post)
  or grant_q    (pre, post)
  or read_fifo  (pre, post)
  or p_critical (pre, post)
  or q_critical (pre, post)
```

END PVS

Note that the problem of action scheduling is completely avoided. During proof, we must split our proof once for each action, deciding whether or not it actually executes. For our traffic signals example, this yields 2^{10} (1024) individual theories that would need to be checked.

There are a number of problems with this. First, the exponential explosion of proof obligations swiftly overwhelms PVS if we try to prove anything larger than a toy example. Second, we have a composability problem. In order to test properties over multiple clock cycles, we need to apply this disjunctive transition predicate multiple times. In the case of this example, this would mean 2^{10n}, where n is the number of clock cycles we are simulating. If we needed to simulate four consecutive clock cycles, which is a reasonable thing to ask, we are looking at 1.1×10^{12} proof obligations. This is simply not feasible, due to time and memory constraints.

Our algorithm may have seemed overkill, but the fact is that the Bluespec scheduler is deterministic, so of the 1024 theorems generated via the monadic method, only one of them is true to the eventual generated hardware. It is a non-trivial amount of work to accurately simulate the scheduler, but doing so allows us to make a deterministic decision about which rules actually fire, and we therefore avoid the composability problem.

4.3 Optimizations Addressing Scalability

One of the largest problems with the scheme described above is the generation of two subtrees at each node of the universal scheduling tree. This yields an algorithmic complexity of $O(2^d)$, where d is the depth of the tree (maximally the number of actions to be scheduled). Even moderately sized BSV descriptions quickly run into projected runtimes of thousands of years, and translating a large design (such as the RapidIO subsystem) would consume all computer memory currently existing (approx. 295 Exabytes (Nguyen, 2011)) and still not be satisfied by many orders of magnitude. This is commonly known as the state explosion problem.

To address these limitations, several algorithmic techniques were applied. This was necessary in order that the RapidIO subsystem could be usefully translated and operated on by PVS.

4.3.1 Tree Simplification via If Expression Observations

One immediate simplification involves state specific trees. This optimization consists of a fairly rudimentary traversing and refactoring of the tree according to the following properties of if-then-else expressions:

$$\text{if } True \text{ then } p \text{ else } q = p$$

$$\text{if } False \text{ then } p \text{ else } q = q$$

$$\text{if } b \text{ then } p \text{ else } p = p$$

The application of the above rules had the effect over medium designs of reducing resultant PVS files from hundreds of thousands of lines of code to under one thousand in most cases. Designs could then be successfully typechecked by PVS in a time-frame of minutes, rather than days or weeks, with no loss to semantic integrity.

4.3.2 Module Hierarchy Action Set Refinement

As the designs being translated grew, so too did their organization into a hierarchy of modules. The naïve approach creates one universal schedule for all actions contained in all modules. While this is a technically correct approach, as states within sibling modules will never conflict with one another, and therefore their relative order of execution is immaterial, it is far from an optimal one. In the file `RapidIO_MainCore.bsv`, the `MainCore` module collected 84 actions, giving the binary tree a depth of 84 action decisions (a tree with 1.93×10^{25} nodes). Obviously, this was a long way from an acceptable state of affairs.

We observe, however, that the state elements of any particular module may only be written to by the module itself (through statements), or by any of the module's supermodules (via method calls). As such, the state specific schedules of any particular module need only concern itself with the actions in the above specified modules.

Thus, a new recursive hierarchy was introduced which corresponds with the module hierarchy of the underlying BSV modules. Each node in this new tree contains a "universal" tree applicable only to the module which the node corresponds to. These trees are generated from subsets of the set of all actions, i.e., only those actions in the module itself or its supermodules.

This scheme sufficed to simplify to a manageable size the majority of the state trees in the module hierarchy under examination in §6.5, which attempts the translation of the entirety of the Shakti processor's implementation of the RapidIO packet processing subsystem. However, on the deepest branch of the tree, action depths were found of up to 42 actions (a tree with 4.40×10^{12} nodes). In particular, the outgoing packet concatenation system is large, both in terms of the number of modules in the hierarchy, and the number of actions per module. This method of simplification is most effective on wide, shallow hierarchies, and is less effective on deep hierarchies. The RapidIO implementation is a combination of shallow and deep hierarchies, so this method only takes us so far in addressing the scalability problem. While a valuable addition to the optimization scheme, other optimizations were required to achieve scalability. This is discussed in the next section.

4.3.3 Action Merger via Schedule Independence Checking

Continuing the optimization effort described above, a new structural observation was necessary to further simplify the decision tree. This observation derives from the scheduling interchangeability of actions which neither conflict with, nor pre-empt each other. If two actions meet the following conditions, they may be merged into one action:

- The actions must conflict with the same set of actions
- The actions must preempt the same set of actions
- The actions must be preempted by the same set of actions.
- The actions must have identical guard expressions.

Essentially, if a group of actions have no scheduling differences between them, and are guarded by the same expression, they will either all be executed or none will be executed (atomicity) and their order of execution is immaterial (parallelism). As such, these actions may be merged into one action. This action would be guarded by the common guard expression, and the statements would be the set of all the statements of the merged actions. The statements themselves would not be merged or altered in any way, just collected together into a single action.

This has the net effect of eliminating scheduling redundancy, and leaving behind the set of actions with *unique* scheduling properties. This ties runtime complexity to the path complexity of the state element calculations, rather than to the abstraction and modularization techniques implemented by the designer in the interests of readability and maintainability. By optimizing the RapidIO library in this manner, a maximum recursion depth of nine was found (a tree with 512 nodes), bringing even the most complex call paths of the design within the realm of that which can be typechecked in a practical amount of time.

4.3.4 Top Level Methods and Schedule Indexing

Methods are the only mechanism for input/output of BSV modules, and the root module of a hierarchy is no exception. In order to pass data into a module, invocation of methods is necessary, but data may be read from modules for free, as read operations have no scheduling implications for register based memory. As will be discussed in §5, the user of the translation algorithm is expected to have sufficient knowledge of the design under verification to know which methods must be invoked to demonstrate the behaviour under examination. As such, a mechanism is provided to specify which sets of methods they would like schedules generated for.

In the top-level PVS file generated by BAPIP, there is a listing of the generated schedules, the methods they invoke, and an index number. This index number must be used as the first argument of any output method or transition predicate that is invoked by the user. An earlier approach to this problem, (enumeration of every possible method schedule), is completely infeasible for even moderately sized designs. The number of possible schedules is 2^n, where n is the number of action methods, as, for each method, that method may be invoked or not. The top level file of RapidIO alone contains 64 such methods. Fortunately, very few possible schedules are interesting, and those which are interesting may be selected prior to translation.

The general procedure is the selection by the user of those methods which the user wishes to generate transitions for. These method selections must then be composed into an auxiliary file by the user, to be read by the translation software at translation time. It is not necessary to specify any ordering of method calls, as these are composed within the PVS proof sequent.

Our running example above is not a good example of this, due to the low number of methods available. In general, though, in order to generate a state transition predicate that uses the method M1, a file that indicates such must be provided in advance. The exact manner of the composition of these files is detailed in §A.4.1.

4.4 Conclusion

In this section, our core algorithm has been explained in detail, using running examples. We have discussed the ways in which our algorithm constitutes an improvement over the primary ancestor of our work, Richards and Lester (2011), and we have discussed methods employed in improving the scaleability of what is unfortunately an exponential algorithm. This approach is necessary to provide a deterministic solution to the multiple clock cycle, and therefore the real-time semantics of BSV.

5. PROVING THE CORRECTNESS OF BSV IMPLEMENTATIONS

While the extraction of logical models from BSV descriptions and their subsequent embedding in the higher order logic of the PVS proof system is fully automated, the generation of proofs of correctness cannot be fully automated. A greater degree of automation in proof generation than is currently presented is theoretically possible, particularly if the requirements are specified in a format which is easily digestible to PVS. This is, however, left as future work. The proof process in general requires more information than is contained by the BSV description under examination, most notably, some formalization of the functional requirements of the module under examination. It is impossible to verify that a module implements a set of requirements without the availability of those requirements. As such, §5 will describe the methods by which the PVS output of the translator may be used to generate proofs of correctness.

5.1 Refinement via Timing Simulation

The method for theorem construction presented in the previous section, while having the advantage of simplicity, does not address the problems that arise from attempting to make statements about the time it takes a module to perform some desired function. The strict timing requirements of safety-critical applications make this a highly desirable capability. Fortunately the methodology presented above may be refined by adapting the methods of Wassyng et al. (Wassyng et al., 2005), in particular, the `tick` type and supporting theories.

These libraries introduce a simple modal logic, making available a discrete timeline which may be traversed by various library functions. By reformulating key variables, such as the state variable, into functions accepting a value of type `tick` (i.e., some point on the timeline) and returning its normal type, we introduce a concept of time to our variables, so that they may have different values at different points on the timeline. This is particularly useful in the state type passed to transition predicates. Where before we had separate, individual variables for pre and post state, we have now arranged the progression of states into a a timeline, from which various points are accessible via either direct addressing or modal functions such as `next(s)`. As such, we may now quantify over time as well as input values, allowing us to assert a whole new set of antecedents, such as a transition predicate holding for all points on the timeline, though no case study is presented which illustrates this capability.

In order to use the `tick` type, it is necessary to parameterize the top-level theory with the duration of each clock tick, the `delta_t`. When provided with the clock period, this allows us to relate intervals on our timeline to real time, giving us a mathematical foundation for claims regarding the real-world timing behaviour of BSV modules.

Use of the `tick` type is also necessary for the interaction of the work presented herein with the formalized tabular specifications provided by Pang et al. (Pang et al., 2015).

5.2 *Using Tabular Specifications to Construct Theorems*

While it is entirely possible to construct theorem antecedents and consequents manually from requirements specifications, it would be better to integrate these specifications with as little manual intervention as possible. In order to do so, it is first necessary to have requirements specifications encoded in a format parsable for PVS. Fortunately, in the case of the IEC 61131-3 function block library (IEC, 2013), we may rely on the work of Pang (Pang et al., 2015), which not only formalizes the IEC 61131-3 function blocks, but encodes those formal tabular specifications in PVS. The Pang tabular expressions are accompanied by proofs of completeness, disjointness and consistency.

Similarly to the transition predicates produced by the BAPIP translation tool, the tabular specifications presented by Pang are also predicates, but unlike our transition predicates, they make no attempt to specify internal behaviours. For an example, please see the code listing below. Tabular expressions also only concern themselves with one output variable at a time, so (with regard to sequent construction) modules with multiple output variables must invoke one tabular expression for each output variable, as opposed to BAPIP transition predicates, which calculate all output values in one invocation. The tabular specifications are encoded as tests of equality between an output variable and a value calculated from input variables and previous output variables via a condition table, as shown below. As such, they readily integrate into the established sequent structure. However, we must ensure input and output variables are handled correctly.

PVS

```
f_q_val (qh,ql)(t): bool =
    TABLE
     | qh(t) OR ql(t)           | True  ||
     | NOT qh(t) & NOT ql(t) | False ||
    ENDTABLE
```

END PVS

In the above code snippet, we declare a function returning a function on three parameters mapping to the Booleans. The two parameters `qh` and `ql` are also functions resulting in Booleans, parameterized by a timeline position, and are inputs supplied to the hardware module by methods. The `(t)` phrase indicate the variable parameterizing `qh` and `ql`. Normal `TABLE` notation is used to map conditions to truth values. In this particular case, it would be trivial to simply state the disjunction of the two inputs as the output, but the methods applied here are useful for larger, less trivial tabular expressions.

In a previously published work (Moore and Lawford, 2017), it is stated that some minor changes to the previously published tabular specifications were required in order to use them with the BAPIP transition predicates. In particular, it was thought necessary to adjust the time at which the tabular specification specifies its result. Prior to modification, tabular specifications describing combinational hardware behaviours had produced a result in the same clock cycle as the inputs were received. This was due mainly to the limitations of the BAPIP software at the time of the paper's publishing, In particular, the "wire" datatype had not been implemented, preventing the transport of data between actions in a single cycle. In this manner, the tabular specification predicates have been modified to compare the value calculated via condition table to the following clock cycle's output variable. This approach has been made obsolete by expansion of the translator's capability.

In terms of theorem construction, tabular specification predicates are included in precisely the same manner as BAPIP transition predicates, but without pre and post condition expressions. Both the specification predicate and the transition predicate can accept the same input variables. For the Pang predicates, the time at which the table is applied is specified at the end of the argument list, whereas in the transition predicate each individual input value must be provided with the time as an argument, within the argument list (see below). The state function should be provided in a similar manner, with the state value at time `t` being given as the pre-state, and the state value at time `t+delta_t` or `next(t)` being given as the post-state. The specification table has no such requirement, but an output variable must be pre-declared. Pre-conditions are automatically provided by the tabular specification, and post-conditions take the form of a test of equality between the output variable of the tabular specification and the relevant method call applied to the relevant time of the state function, as given below. This theorem is derived from our earlier work (Moore and Lawford, 2017). Though an updated version is presented in §6.1, this sample has been selected because it illustrates the concept of antecedent and consequent logical expressions in a more direct manner.

PVS

```
line1_2 : theorem LIMITS_ALARM_t_set_Alarms
  ( s(pre(t))   % Pre-state record
  , s(t)        % Post-state record
  , xin(t)      % Input under test
  , hi(t)       % High Threshold
  , lo(t)       % Low Threshold
  , ep(t)       % Epsilon
  )
  and (ep(t) > 0)
  and (xin(t) >= (hi(t) - 2*floor(div(ep(t),2))))
  and (xin(t) <= hi(t))
    implies LIMITS_ALARM_get_qh(s(pre(t)))
      = LIMITS_ALARM_get_qh(s(t))
```

END PVS

In this particular example, `pre(t)` is being used instead of `next(t)`. This has to do with the data path represented by this module, which will be discussed in greater detail in §6.1. In effect, we are checking to see that, for the same input, both the specification and the implementation produce the same output at the same time. If successfully proven, we may say that the Bluespec module implements the requirements.

It is sometimes necessary to add to the antecedents a premise equating the history of the tabular specification's output variable to the history of the transition predicate's state. This is necessary when the module is not purely combinational, relying on the previous value of some result to calculate the next. Without such a statement, PVS would check for equivalence in cases where both mechanisms had different histories. It is our position that the assumption that both mechanisms have the same history is reasonable. A deductive approach is taken to this, rather than an inductive proof starting from equivalent initial conditions, because it seemed the simpler approach. It is highly probable that the techniques discussed here could be make far more efficient via the application of more advanced proof techniques.

5.3 Proofs of Consistency

In order to validate a module, it is insufficient merely to demonstrate that it meets its functional requirements. It is also necessary to demonstrate that the antecedents of our proof sequent are not vacuous. If antecedents are self-contradictory, it is possible to deduce any possible outcome. This means, for example, that it would be possible to deduce some desired consequent from a set of antecedents, *and* the Boolean negation of the same consequent. In such a case, any demonstrated result is meaningless. As such, we must demonstrate the *consistency* of our theorems, where the term consistency is used in the same manner as (Camilleri et al., 1986). In short, we must demonstrate that, for every pre state there exists a valid post state, even if the relation is reflexive.

In order to demonstrate the consistency of our automatically extracted transition predicates, we must demonstrate that they restrict the space of possible post-states without eliminating it. In other words, for all pre-states and combinations of input, there must exist at least one post-state, otherwise there would exist some combination of inputs and state that would cause the transition predicate to always return false. This would introduce a contradiction as a premise, allowing us to prove any consequent. This is another way of saying this is that our transition predicates must be *left-total*, a property discussed more fully in §2.1.3.

Constructing a theorem to evaluate this is straightforward. The theorem must state that for all pre-states and input variables, there must exist a post state such that the transition predicate holds. This theorem necessarily does not take the antecedent-consequent form of the theorems testing functionality.

```
Consistency : Theorem
  FORALL pre, <input variables> :
    EXISTS post :
      transition_predicate (pre, post, <input variables>)
```

For a description of how to discharge these proofs, and the manner of their automatic generation, see 2.2.6.

6. CASE STUDIES

In this chapter we will examine five case studies, which serve as demonstrations for the BAPIP translation process, as well as the derivation of associated proofs. The case studies are as follows:

1. The Limits Alarm Function Block — A very simple example, previously published (Moore and Lawford, 2017).
2. The Alarm Int Function Block — A simple example demonstrating the translator's TSP2BSV and TSP2PVS modes.
3. RapidIO Read/Write Size and Word Pointer Decoder Module — A more complex example demonstrating the ability to verify BSV functions.
4. RapidIO Read Size and Word Pointer Encoder Module — The most complex working example, demonstrates action arbitration and wire handling.
5. Progress Towards RapidIO Transaction ID Echoing — Theoretical groundwork for a highly complex example, encompassing most of the RapidIO subsystem under examination.

All case studies have associated code listings in the appendix of this thesis.

6.0.1 Introduction to RISC-V and RapidIO

Three of the case studies provided in this work (§6.3, §6.4 & 6.5) are applied to a Bluespec SystemVerilog implementation of the RapidIO message packet passing subsystem of the RISC-V processor Instruction Set Architecture (ISA) available in the Shakti processors project, by (Gala et al., 2016). It is therefore appropriate to introduce the RISC-V processor, the Shakti project, and provide an overview of the RapidIO framework.

RISC-V

RISC-V is an open-source specification for the design of computer processors (Porter III, 2018). RISC stands for Reduced Instruction Set Computer, and RISC-V is an open-source hardware initiative led by the RISC-V Foundation (2020). In order to understand the purpose of RISC-V, one must consider the importance of ISAs in the design of computers. Processing, as performed in Von Neumann architectures, comprises the manipulation of memory through the execution of instructions. The set of instructions a processor implements, and the manner of their implementation, are the objects of a great deal of design effort on the part of processor manufacturers. Each instruction that is included must have corresponding electronic circuitry within the final processor. The set of instructions that a processor implements is that processor's ISA, and there are multiple ISAs used by different manufacturers, for example, Intel's ubiquitous x86-64, or Zilog's Z80, which was highly popular in embedded systems applications in the 1980s, such as audio synthesizers and arcade machines. CISCs, or Complex Instruction Set Computers, often have large numbers of instructions. This is sometimes due to the need for backwards compatibility with legacy software. One of the primary design goals of the RISC-V architecture (and all other Reduced Instruction Set Computer architectures, of which there are many) is to implement a minimized set of instructions, decreasing the size, cost, and design effort required to implement the processor. (Porter III, 2018) There is a trade-off between the number of instructions in an ISA and the amount of memory the controller requires. Reducing the number of instructions a CPU can execute necessary reduces the expressivity of the associated machine language. Given some complex operation, a RISC must use at least as many instructions as a CISC, and possibly many more, because limiting the number of instructions limits the expressivity of the assembly language. Not only do more instructions take more CPU cycles to execute, but the program itself will tend to be larger, requiring more instruction memory. In the early days of computing, when memory was expensive, the economic incentive was to build up the CPU so that less memory would be required. As memory has become less expensive and CPU speed has increased over the decades, the RISC approach is generally favoured, and now Intel and AMD are supporting CISC ISAs with RISC-style micro-instructions (Isen et al., 2009).

Another point of difference between RISC-V and other ISAs is that it is an open-source specification. For many processor manufacturers, precise details about the implementation of specific instructions are a closely guarded secret, protected by intellectual property law, licensing fees and non-disclosure agreements. If this were not the case, any company could manufacture chips based on the x86-64 ISA, for example, and create undesirable competition for Intel. However, many companies and individuals are now recognizing open-source licensing as a viable alternative to this standard industrial practice. Under open-source these designs are much easier to use and contribute to, meaning the development of the design can leverage a far greater pool of talent than any one company could hope to, irrespective of their size and respectability. As such, RISC-V makes itself much more attractive to researchers and industrialists, who can engage with the ISA without incurring prohibitive licensing costs, producing implementations and improvements. This creates an "ecosystem" of RISC-V processors, rather than a handful of strictly regulated sources.

The Shakti Project

The Shakti project is a family of implementations of the RISC-V ISA by (George et al., 2018), and the RISE group at IIT Madras. The work is currently ongoing, and individual cores within the family have different design focuses, such as targeting embedded, control, and mobile processor applications, and security-focused and fault tolerant variants. The Shakti family of processors have been designed and implemented in Bluespec SystemVerilog, citing a higher level of abstraction, "superior behavioural semantics," architectural transparency and parameterizability as justification. Bluespec Inc. has itself recently pivoted towards being a supplier of RISC-V technologies (Bluespec Inc., 2019). Additionally, Bluespec has seen an open-source release on github (`https://github.com/B-Lang-org/bsc`), with it's first open source release in July of 2021.

Because Shakti is an open source project, the source code for these processors and their subcomponents is fully available for viewing and download (Madhusudan, 2018). As such, the BSV designs were available for use as verification case studies. The combination of open source code and the free availability of the standards from which it was derived, as well as the non-trivial size of the example, made Shakti ideal for a demonstration of our techniques.

The RapidIO Interconnect Framework

The particular subsystem focused on in our case studies is the RapidIO message passing system. Generally speaking, the various chips of a circuit board require some sort of framework for the transmission of information. There are many approaches to this problem; the one used by Shakti is RapidIO, an open standard.

The RapidIO system passes information through its network via message packets. These packets often represent instructions to be carried out or requests for data. The format of the message packets is rigidly determined by the specification (RapidIO.org, 2017), and so is the format of response packets.

The RapidIO subsystem under examination is that which forms response packets in response to packets received from the network. The focus of the verification case studies presented in this thesis, at a high level, is to verify that the response packets formulated by the Shakti BSV implementation of RapidIO are consistent with various properties derived from examination of the RapidIO specification.

6.1 The Limits Alarm Function Block

This case study serves to illustrate, via a simple example, the process of verification using BAPIP and PVS. It uses a BSV file created by the author from specifications found in (IEC, 2013), translates the file into PVS using BAPIP in BSV2PVS mode, and then verifies consistency with tabular specifications set out in (Pang et al., 2015). This hardware module was first presented as a case study in (Moore and Lawford, 2017), but has been modified since. These modifications take into account the fundamental restructuring of the translator which has occurred since 2017, and uses a much less convoluted proof scheme.

6.1.1 The Hysteresis Function Block

In order to understand the Limits Alarm function block, it is first necessary to understand the function block Limits Alarm is mainly constructed from, the Hysteresis block.

Hysteresis is a general principle of control systems which seeks to prevent rapid oscillation between two states. The classic example is a home's heating system. In this system, the inputs to the system are the temperature reading from a thermal sensor, and a temperature set-point from a thermostat connected to the home heating system. The output is the on or off state of the home's furnace. In a naive implementation of this control system, one would simply turn the furnace on when the temperature reading is less than the set-point, and off when it is greater than the set-point. There is a problem with this approach, however. We have set up a system where the room temperature will oscillate around the set-point, with a frequency depending on how long the furnace takes to start effecting the room temperature. If this oscillation is rapid enough, it can be damaging to the electrical and mechanical components of the system. This effect can be greatly magnified by a noisy temperature signal.

The solution to this problem is to enforce a dead band around the set-point. That is, we pick some reasonable error value ϵ, and make it so that the system's output does not change while inside the dead band. So, in our furnace example, if we set our home to $20°C$, and decided on an epsilon of $1°C$, we would see behaviour in Figure 6.1. The room's temperature fluctuation has been modelled as sinusoidal, not to accurately describe the behaviour of a physical system, but as a stand-in for a more general class of oscillations.

The International Electrotechnical Commission's standard on PLC programming languages (IEC, 2013) implements hysteresis behaviour in the Structured Text (ST) PLC language as described by Figure 6.2. This diagram was first published in (Pang et al., 2015), though it is a direct republication of (IEC, 2013).

In order to verify the implementation of the Hysteresis block given in (IEC, 2013), or any other implementation, it was necessary to formalize the specification into a more rigorous form. This was undertaken by (Pang et al., 2015), and resulted in the tabular specification given in Figure 6.3. Here and elsewhere, "NC" is an abbreviation for "No Change", where the output variable retains its previous value.

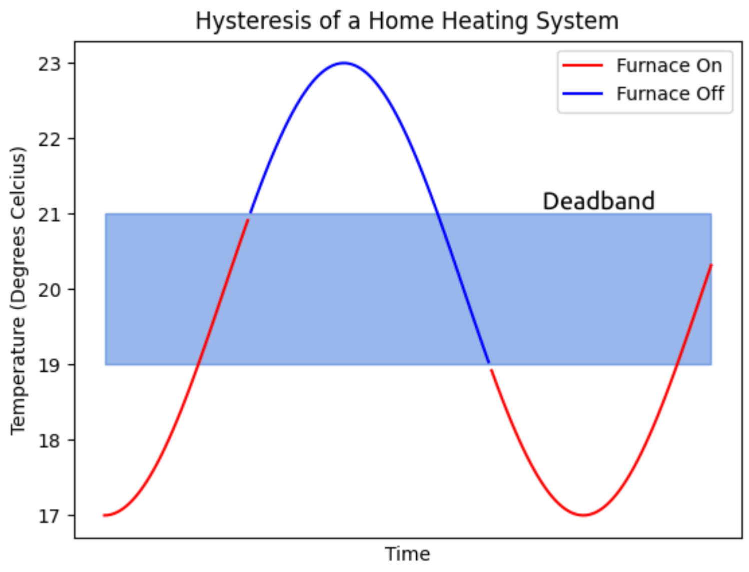
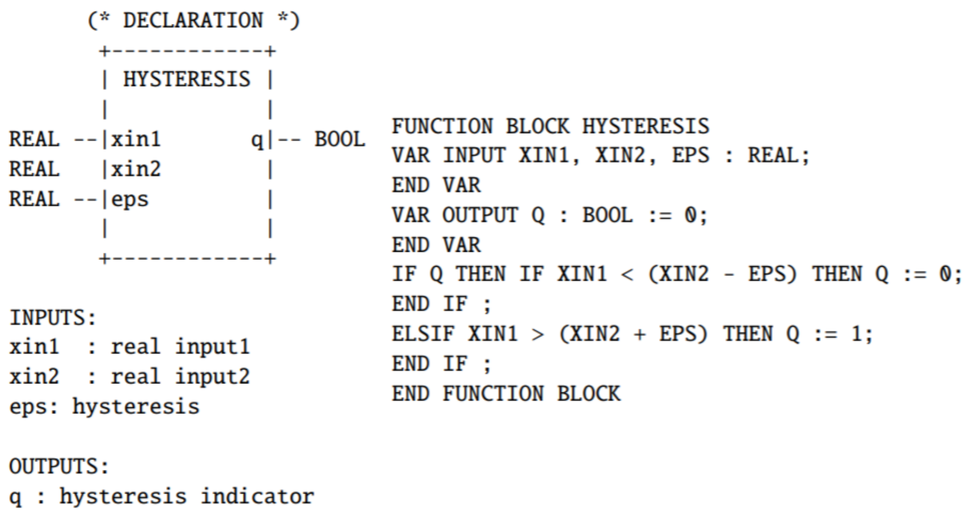


Figure 6.1: Hysteresis behaviour of a home furnace



```
(* DECLARATION *)
 +------------+
 | HYSTERESIS |
 |            |
REAL --|xin1       q|-- BOOL
REAL   |xin2        |
REAL --|eps         |
       |            |
       +------------+

INPUTS:
xin1  : real input1
xin2  : real input2
eps: hysteresis

OUTPUTS:
q : hysteresis indicator
```

```
FUNCTION BLOCK HYSTERESIS
VAR INPUT XIN1, XIN2, EPS : REAL;
END VAR
VAR OUTPUT Q : BOOL := 0;
END VAR
IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0;
END IF ;
ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1;
END IF ;
END FUNCTION BLOCK
```

Figure 6.2: Implementation in STL of Hysteresis Block (Pang et al., 2015)

	Result
Condition	q
xin1 $<$ (xin2 $-$ eps)	false
(xin2 $-$ eps) $\leq$ xin $\leq$ (xin2 $+$ eps)	NC
xin1 $>$ (xin2 $+$ eps)	true

assume: $eps > 0$

Figure 6.3: Hysteresis behaviour of a home furnace

From here, the following BSV implementation was produced by the author.

BSV

```
package HYSTERESIS;

interface HYSTERESIS;
method ActionValue#(Bool) set_Inputs
  ( Int#(16) xin1
  , Int#(16) xin2
  , Int#(16) eps
  );
method Bool get_q();
endinterface

module mkHYSTERESIS (HYSTERESIS);
Reg#(Bool) q <- mkReg(False);

method Action set_Inputs (xin1, xin2, eps);
  q <=   (q && (((xin2-eps)<=xin1) && (xin1<=(xin2+eps))))
        ||  (xin1>(xin2+eps)) ;
endmethod

method Bool get_q ();
  return q;
endmethod

endmodule : mkHYSTERESIS
```

```
endpackage
```

END BSV

6.1.2 The Limits Alarm Function Block

Limits Alarm is a variant type of the Hysteresis block which triggers an alarm if the value being tested exceeds either an upper or a lower limit. It is composed of two hysteresis blocks, one which tests the upper threshold, and another which tests the lower threshold. A global alarm signal will activate if either hysteresis block indicates that the value under test has exceeded a threshold. The Limits Alarm block has three corresponding output signals for the high threshold (QH), the low threshold (QL), and the global alarm signal (Q). Tabular specifications for these three signals are presented in Figure 6.4 (Pang et al., 2013, 2015).

Condition	**QH**
$X > H$	True
$(H - EPS) \leq X \leq H$	No Change
$X < (H - EPS)$	False

Condition	**QL**
$X < L$	True
$L \leq X \leq (L + EPS)$	No Change
$X > (L + EPS)$	False

Condition	**Q**
$QL \vee QH$	True
$\neg(QL \vee QH)$	False

assuming $(EPS/2) > 0$

Figure 6.4: Limits Alarm Tabular Specifications for QH, QL, and Q outputs

Here, X is the signal under test, H is the high threshold, L is the low threshold, and EPS is the width of the hysteresis dead band. A full code listing for this block may be found in §D.1.1.

Condition	**QH**
$X > H$	True
$(H - (EPS/2) - (EPS/2)) \leq X \leq H$	NC
$X < (H - (EPS/2) - (EPS/2))$	False

assuming $(EPS/2) > 0$

Figure 6.5: Limits Alarm Tabular Specification for Integer Variant of QH

Some concessions to implementation were made during the development of the correctness proof for this block, further specifying (IEC, 2013). The integer implementation of this block highlights a disparity between the way Limits Alarm and Hysteresis handle thresholding. In the case of Hysteresis, the dead band is $X \pm EPS$, with $X + EPS$ being the upper limit, and $X - EPS$ being the lower. In Limits Alarm, H is the upper limit of the upper threshold, and the lower limit of the upper threshold is $H - EPS$. In order to use a Hysteresis block to produce this effect, the threshold value it is given must be $H - EPS/2$, critically introducing division operators. For real data types this poses no particular threat, as real values can (theoretically) be divided perfectly, but integer division is not the same operation. In order to prove our integer implementation, it was necessary to revise our requirements such that they reflected this. The tabular specification in Figure 6.5 illustrates the changes that were necessary.

Essentially, during the construction of the QH function block, the term $H-EPS$ is actually an algebraic simplification of collection of $H-(EPS/2)-(EPS/2)$. Under normal mathematical conditions this simplification is valid, as $X/2 + X/2 = X$. However, under integer arithmetic, which contains an implicit floor operation, this equality will only hold for even values of X. Therefore, to accurately reproduce the integer arithmetic behaviour of the hysteresis block, it was necessary to reverse this algebraic simplification.

Another design decision made was the clock cycle on which the global alarm Q was available. The global alarm Q, from a black box perspective, must be available on the clock cycle following the setting of the module's inputs via the input method. As expressed by Pang et al. in PVS (Pang et al., 2015), the Q tabular specification requires values of QH and QL as inputs. Since these are calculated values, and not inputs to the module, they do not become available until one clock cycle has elapsed. Up to this point, it has been typical to rewrite the tabular specifications slightly such that they apply to the output value on the following clock cycle. For Q, the tabular specification applies to the same clock cycle that it's inputs occur in. In the implementation, this is achieved by allowing the access method for Q to directly access QH and QL, calculate Q, and return that value. This bypasses the need to calculate and store the value of Q within the Limits Alarm module itself.

6.1.3 Constructing the Proof Sequent

For this block's proof of correctness, our first step is to encode the tabular specifications given above in PVS, as follows.

PVS

```
t: VAR tick

s : VAR [tick −> LIMITS_ALARM]
pre, post, LIMITS_ALARM_var : VAR LIMITS_ALARM

%%% Requirements Tables

x, l, h, eps: Var Int(16)
qh, ql, prev : Var bool

qh_req (x,h,eps,prev) : bool =
  TABLE
  |    x > h                              | True ||
  |    x >= (h−div(eps,2)−div(eps,2))
  AND x <= (h−div(eps,2)+div(eps,2)) | prev ||
  |    x <  (h−div(eps,2)−div(eps,2)) | False ||
  ENDTABLE

ql_req (x,l,eps,prev) : bool =
  TABLE
  |    x < l                              | True ||
  |    x <= (l+div(eps,2)+div(eps,2))
  AND x >= (l−div(eps,2)+div(eps,2)) | prev ||
  |    x >  (l+div(eps,2)+div(eps,2)) | False ||
  ENDTABLE

q_req (qh, ql) : bool =
  TABLE
  | qh OR ql             | True  ||
  | NOT qh AND NOT ql | False ||
  ENDTABLE
```

END PVS

Requirements tables are here expressed as functions on all the parameters used in the table, to the type of the variable to which the table applies. In this case, we are including the previous value of `qh` and `ql` as a parameter, to avoid having to define the table recursively. Otherwise, these tables should appear as relatively direct implementations of the requirements tables given in Figure 6.4

We may now construct the correctness theorem itself.

PVS

```
correctness : theorem
 forall (X:Int(16), H:Int(16), L:Int(16), EPS:Int(16)):
      transition (1,s(0),s(1), X, H, L, EPS)
  and div(EPS, 2) > 0
   implies
        qh_req(X, H, EPS, s(0)`high_alarm`q)
          = get_qh(0,s(1),s(1))
    and ql_req(X, L, EPS, s(0)`low_alarm`q)
          = get_ql(0,s(1),s(1))
    and q_req( qh_req(X, H, EPS, s(0)`high_alarm`q)
             , ql_req(X, L, EPS, s(0)`low_alarm`q)
             )
          = get_q(0,s(1),s(1))
```

END PVS

After the declaration of the theorem itself, our next line binds all free variables in the subsequent lines and types them. Next, we invoke the transition predicate, using schedule one, which in this case included the method which sets the inputs from the input parameters provided. This proof hinges on the fact that `X`, `L`, `H` and `EPS` are consistent where they are used. We make sure that the requirements tables get the same inputs as the hardware model by providing the same variable to both. After the transition is invoked, we include the implicit condition that the epsilon value must be greater than zero. This concludes the antecedents.

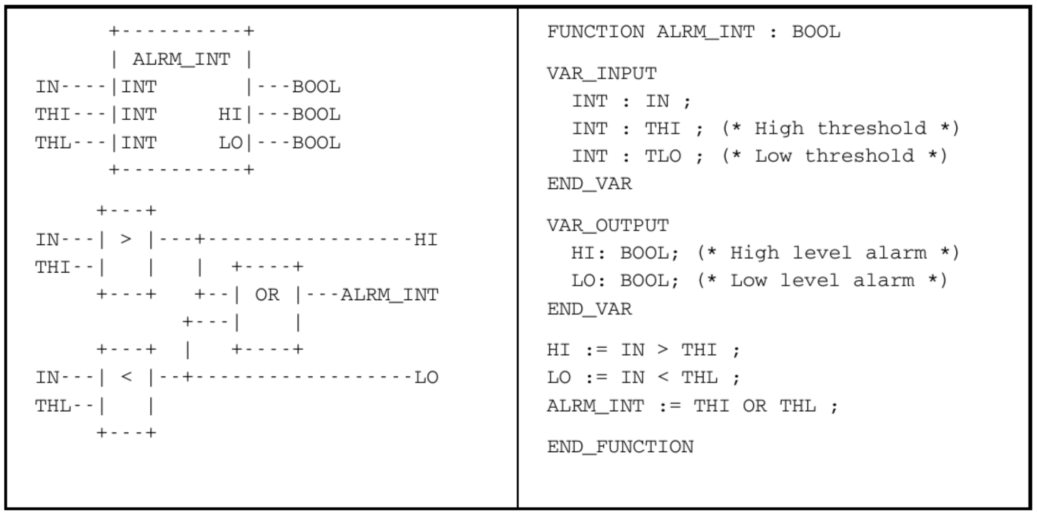
The consequents are somewhat more complex, but still relatively easily understood. Correctness is here defined as simultaneous satisfaction of three equalities, in the post-state of the above given transition. The calculated result of the requirements table `qh_req` must match the result of `get_qh()`, and similarly for `ql` and `q`. Note that value methods must also have a schedule specified. It works in either case for this example, but it is possible for the values written to wires to modify the outputs of methods in some BSV designs, so it is also necessary to tell an output method which input methods have been called in the requested clock cycle. In this case, we specify schedule zero, which indicates that no methods have been invoked this clock cycle.

6.1.4 Proving the Proof Sequent

In order to prove the above sequent, the following steps were taken.

- `(skolem!)` → Skolemization of universally quantified variables.
- `(expand ...)` → Expanded definitions in antecedent
- `(flatten)` → Rewrite sequent so that `implies` becomes turnstile, antecedents come before turnstile, and consequents come after it.
- `(split)` → Generation of one sub-proof for each of the three consequents.
- For each subproof:
  - `(expand ...)` → Expanded definitions in consequent, including requirements tables
  - `(grind)` → At this point, the rest of the proof is completed automatically by a general purpose deductive strategy.

After the application of these proof strategies, the model of our BSV implementation of the Limits Alarm block, as extracted by BAPIP, was proven consistent with the tabular specification presented in Figure 6.4. Discharging all proof obligations takes an average of 0.66s on the author's computer (ASUS TUF506QM, AMD Ryzen 7 5800H, 16GB RAM, running Linux Mint 20.3). All files used in this case study are available in §D.2, or in the BAPIP project github repository: `https://github.com/nmoore771/bapip`.



```
      +----------+
      | ALRM_INT |
IN----|INT       |---BOOL
THI---|INT     HI|---BOOL
THL---|INT     LO|---BOOL
      +----------+

     +---+
IN---| > |---+----------------HI
THI--|   |   |  +----+
     +---+   +--| OR |---ALRM_INT
             +--|    |
     +---+   |  +----+
IN---| < |--+-----------------LO
THL--|   |
     +---+
```

```
FUNCTION ALRM_INT : BOOL

VAR_INPUT
  INT : IN ;
  INT : THI ; (* High threshold *)
  INT : TLO ; (* Low threshold *)
END_VAR

VAR_OUTPUT
  HI: BOOL; (* High level alarm *)
  LO: BOOL; (* Low level alarm *)
END_VAR

HI := IN > THI ;
LO := IN < THL ;
ALRM_INT := THI OR THL ;

END_FUNCTION
```

Figure 6.6: Specifications for ALRM_INT function block, as published in (IEC, 2013)

6.2 *Alarm_Int and Automatic Generation using Tabular Specifications*

The purpose of this case study is to demonstrate the automatic generation of BSV files, PVS encodings, and proofs from a PVS encoded tabular specification.

The integer alarm `ALRM_INT` is a high-low threshold alarm similar to the double hysteresis `LIMITS_ALARM`. It takes as input a high and low threshold value (`thi`, `tlo`), and a test value (`inp`). It has three outputs: a high alarm `hi`, a low alarm `lo`, and a global alarm `alrm_int`. The original specifications for the ALRM_INT block appear in Figure 6.6

Tabular expressions for these values are given in Figure 6.1. While originally appearing in (Pang et al., 2015), they can be reproduced with minimal effort from the specifications given in Figure 6.6

6.2.1 *Encoding in PVS*

When encoding tabular expressions in PVS, the first consideration is whether a particular table describes primary or secondary data. By primary data, we mean data which can be calculated directly from supplied method arguments,

Condition	**hi**
$inp > thi$	True
$inp <= thi$	False

Condition	**lo**
$inp < tlo$	True
$inp >= tlo$	False

Condition	**alrm_int**
$hi \wedge lo$	True
$hi \wedge \neg lo$	True
$\neg hi \wedge lo$	True
$\neg hi \wedge \neg lo$	False

Tab. 6.1: Integer Alarm - Tabular Expressions for *hi*, *lo*, and *alrm_int*

and stored in registers. In hardware terms, input signals are fed into some amount of combinational logic, with the results stored in sequential elements (such as registers). The data in these sequential elements are referred to here as primary data. Outputs from a module may have combinational logic between the sequential components (registers, etc.) and the output data lines. When module outputs are taken from this combinational circuitry, rather than directly from the registers, we refer to these as secondary data. If a requirements table references the results of other requirements tables, it is secondary, and not primary data, as primary data is calculated purely from inputs to the module. In the case of `ALRM_INT`, the only result referencing other table results is `alrm_int`, as neither `hi` nor `lo` reference any variable for which a tabular expression exists. Therefore, `alrm_int` is secondary, and `hi` and `lo` are primary. For the full code listing of the interpretation of these tabular specifications in PVS, please see §E.3.5. Just the tabular specifications themselves are listed below.

PVS

```
ALRM_INT_req_1(inp,thi,hi)(t): bool = hi(next(t)) =
  TABLE
   | inp(t) > thi(t)   | True ||
   | inp(t) <= thi(t) | False ||
  ENDTABLE

ALRM_INT_req_2(inp,tlo,lo)(t): bool = lo(next(t)) =
  TABLE
   | inp(t) < tlo(t)   | True ||
   | inp(t) >= tlo(t) | False ||
  ENDTABLE

ALRM_INT_req_3(hi,lo,alrm_int)(t): bool = alrm_int(t) =
  TABLE
   | hi(t) & lo(t)            | True ||
   | hi(t) & NOT lo(t)        | True ||
   | NOT hi(t) & lo(t)        | True ||
   | NOT hi(t) & NOT lo(t) | False ||
  ENDTABLE
```

END PVS

These, along with some variable declarations and other paraphernalia, form the input file to BAPIP's `tsp2bsv` algorithm.

6.2.2 *Resultant BSV Description*

Execution of `tsp2bsv` on the above data took on average 15.33ms. An abridgement of the resulting BSV file is presented below. The full file is available in §E.2.

BSV

```
Reg#(Bool) hi <− mkReg(False);
Reg#(Bool) lo <− mkReg(False);

method Action set_Inputs(inp_in, tlo_in, thi_in);
  if ( inp_in < tlo_in ) lo <= True;
  else lo <= False;
  if ( inp_in > thi_in ) hi <= True;
  else hi <= False;
endmethod

method Bool get_lo();
  return (lo);
endmethod

method Bool get_hi();
  return (hi);
endmethod

method Bool get_alrm_int();
  if (hi && lo) return (True);
  else if (hi && (!lo)) return (True);
    else if ((!hi) && lo) return (True);
      else return (False);
endmethod
```

END BSV

Some obvious Boolean simplifications can be observed here. First, the entire contents of the `get_alrm_int()` are obviously equivalent to `hi` or `lo`. It is the author's contention that this simplification would make more sense implemented in the tabular specifications themselves. In `TSP2BSV` or `TSP2PVS` modes, the translator attempts to create as faithful and direct an implementation of the tabular specifications as possible. Further, replacing this if expression with a single return statement would require the translator to interpret the data type of the table. For the moment, the translation process is type-agnostic in this respect. These added optimizations are reserved as future work.

Please note the differing treatments of primary and secondary processes. The primary processes calculating `lo` and `hi` are encoded in the `set_Inputs` method, and the results are registered. In contrast, the output variable `alrm_int` is calculated inside the Value method returning the calculated data, via a branching if-statement structure.

Let's for a moment examine this distinction in terms of hardware. During each clock cycle, high and low voltage signals stored in registers are propagated through some combinational circuit. The outputs of that circuit are taken up and stored by the registers for the next clock cycle. These combinational circuits feeding from registers to registers, including any external input signals, are the primary processes referred to above. Output methods in BSV simply tap into the specified registers, and route their signals outside of the hardware module. In BSV, we may interpose additional combinational circuits between the registers and the module output lines. This allows us to calculate results from data stored in registers with no effect on the internal state of the module, the changes made to the signals by these combinational circuits are only visible externally. These are the secondary data referred to above.

It should also be repeated that, because output methods have no impact on module state, there is no need to figure them into the module's scheduling mechanism.

6.2.3 Generation of Proofs of Correctness and Consistency

After having generated a BSV hardware description, BAPIP will continue in `TSP2PVS` mode to generate theorems describing correctness and consistency, and proof tactics that verify them. The contents of the post-translation PVS files may be found in §E.3. The following theorems of consistency and correctness were automatically generated for the `Alrm_int` function block.

PVS

```
%|- mkAlrm_int_REQ : PROOF
%|- (then (grind))
%|- QED
mkAlrm_int_REQ : THEOREM
    transition (1, s(t), s(next(t)), inp(t), tlo(t), thi(t))
  AND ALRM_INT_req_3(hi,lo,alrm_int)(next(t))
  AND ALRM_INT_req_2(inp,tlo,lo)(t)
  AND ALRM_INT_req_1(inp,thi,hi)(t)
  AND (init(t) IMPLIES mkmkAlrm_int(s(t)))
  IMPLIES alrm_int(next(t))
      = get_alrm_int(0, s(next(t)), s(next(t)))
    AND lo(next(t)) = get_lo(0, s(next(t)), s(next(t)))
    AND hi(next(t)) = get_hi(0, s(next(t)), s(next(t)))

%|- consistency_0 : PROOF
%|- (then (skolem!)
%|-  (inst + "transition_val (i!1, pre!1)")
%|-  (rewrite transition)
%|-  (rewrite transition_val)
%|-  (assert))
%|- QED
consistency_0 : Theorem
  FORALL (i : nat, pre : mkAlrm_int) :
    EXISTS (post : mkAlrm_int) : transition(i, pre, post)

%|- consistency_1 : PROOF
%|- (then (skolem!)
%|-  (inst + "transition_val (i!1, pre!1, inp_in!1, tlo_in!1, thi_in!1)")
%|-  (rewrite transition)
%|-  (rewrite transition_val)
%|-  (assert))
%|- QED
consistency_1 : Theorem
  FORALL (i:nat, pre:mkAlrm_int, inp_in_0:Int(16),
      tlo_in_0:Int(16), thi_in_0:Int(16)) :
    EXISTS (post : mkAlrm_int) : transition(i, pre, post,
         inp_in_0, tlo_in_0, thi_in_0)
```

END PVS

`ALRM_INT_Req` on the first line of the overall correctness theorem is the transition predicate containing a call of the method `set_Inputs`, and is asserted. The next three lines assert the three tabular expressions, as encoded in PVS. The final three lines are a conjunction of consequents. Specifically, the assertions that each of the output variables specified in each tabular expression is equal to the results of corresponding method calls from the BSV encoding.

It might be observed from the above theorems that, since t is a timeline, an inductive proof would be the appropriate course of action. We may have expected to prove this theorem for the initial state of the circuit, and then that the theorem will hold for some hypothetical future state `next(t)` (or $t+1$), given that same theorem holding for t. In this case, proof by induction is unnecessary. If the post case can be demonstrated from the pre case without the aid of an induction hypothesis (as is the case above), there is nothing contributed to the proof by adding one. All we would be doing is complicating the proof to no purpose.

The consistency theorem tests for the existence of a valid post state, given either transition predicate generated by BAPIP. Also generated are special functions which serve as the term instantiated by the `prooflite` scripts, also generated by BAPIP, which prove the consistency theorems. These proofs, when taken together, demonstrate the validity of the automatically generated BSV implementation.

Resolution of the correctness theorem took on average 0.52s on the author's computer (ASUS TUF506QM, AMD Ryzen 7 5800H, 16GB RAM, running Linux Mint 20.3), and was discharged by the automatic proof strategy `(grind)`. Proof of the consistency theorems took 0.034s on average. All files used in this case study are available in §E.3, or in the BAPIP project github repository: `https://github.com/nmoore771/bapip`.

6.3 RapidIO Read/Write Size and Word Pointer Decoder Module

The object of this case study is to demonstrate the application of our BAPIP algorithm and associated procedures to a "real life" BSV example. This example was required to be sufficiently complex to constitute a real test of the BAPIP system, and needed to have industry application. Preferably, this example would have a specification document available which, if not sufficiently formal in itself, might be formalized by the authors. Fortunately, we found the Shakti processor project (Menon et al., 2017), which uses the RISC-V standard (Waterman and Asanović, 2017).

6.3.1 Formalization of the RISC-V Specification

The RapidIO standard (RapidIO.org, 2017) falls short of *formal* mathematical rigour in many aspects. Specifications are often encoded in natural language which is insufficient to formally describe the lower-level subcomponents.

Fortunately, this is less the case with Tables 4-3 and 4-4 of the RISC-V RapidIO specification (Waterman and Asanović, 2017). These tables define `Byte_Lanes`, a variable which determines which bytes of a particular chunk of memory are being read from or written to during a memory transaction. This is controlled by two variables, `wdptr` (word pointer) and `rdsize/wrsize` (read/write size). The relationship between these variables is presented in Table 6.3. To illustrate the byte lanes better, values are provided in binary.

wdptr	rdsize	Bytes	Byte Lanes	wdptr	rdsize	Bytes	Byte Lanes
0b0	0b0000	1	0b10000000	0b0	0b1000	4	0b11110000
0b0	0b0001	1	0b01000000	0b1	0b1000	4	0b00001111
0b0	0b0010	1	0b00100000	0b0	0b1001	6	0b11111100
0b0	0b0011	1	0b00010000	0b1	0b1001	6	0b00111111
0b1	0b0000	1	0b00001000	0b0	0b1010	7	0b11111110
0b1	0b0001	1	0b00000100	0b1	0b1010	7	0b01111111
0b1	0b0010	1	0b00000010	0b0	0b1011	8	0b11111111
0b1	0b0011	1	0b00000001	0b1	0b1011	16	
0b0	0b0100	2	0b11000000	0b0	0b1100	32	
0b0	0b0101	3	0b11100000	0b1	0b1100	64	
0b0	0b0110	2	0b00110000	0b0	0b1101	96	
0b0	0b0111	5	0b11111000	0b1	0b1101	128	
0b1	0b0100	2	0b00001100	0b0	0b1110	160	
0b1	0b0101	3	0b00000111	0b1	0b1110	192	
0b1	0b0110	2	0b00000011	0b0	0b1111	224	
0b1	0b0111	5	0b00011111	0b1	0b1111	256	

Tab. 6.2: Table 4-3 from (RapidIO.org, 2017)

The first and most prominent problem with these tables is the failure to distinguish between input and output variables. Contextually, it can be understood that the first two columns are input, and the last two outputs, but this should be more clearly defined. The sequence of the output variable is also given organizational priority over the sequence of the inputs, where the reverse would be a more readable and less error-prone approach. Furthermore, after a certain point in the table, outputs cease to be defined. It might be assumed from the behaviour of the other output variable `Number of Bytes` that subsequent table entries are assumed to repeat the final entry (`0b11111111`), but it is equally possible that these values of the output variable no longer make sense in the context of the input variables. For the purposes of formalizing this specification for use in formal verification, we have assumed blank output entries represent a "don't care" state, where any output will satisfy the specification.

The following code listing contains the formalized specification table. Discounting the "Number of Bytes" output parameter, the read and write operation byte lane decodings are the same. Therefore, one requirements table was generated for both operations. Here, we can see that values have been translated from binary representations into integer representations.

PVS

```
DC(n:nat) : Bit(8)

req_ByteEnable
 ( wdptr : Bit(1) % Word Pointer
 , rwsize : Bit (4) % Read/Write Size
 ) : Bit(8) =
TABLE
                   |[ wdptr = 0 | wdptr = 1 ]|
  | rwsize = 0 |      128     |       8     ||
  | rwsize = 1 |       64     |       4     ||
  | rwsize = 2 |       32     |       2     ||
  | rwsize = 3 |       16     |       1     ||
  | rwsize = 4 |      192     |      12     ||
  | rwsize = 5 |      224     |       7     ||
  | rwsize = 6 |       48     |       3     ||
  | rwsize = 7 |      248     |      31     ||
  | rwsize = 8 |      240     |      15     ||
  | rwsize = 9 |      252     |      63     ||
  | rwsize = 10 |   254   |   127   ||
  | rwsize = 11 |   255   |  DC(0)  ||
  | rwsize = 12 |  DC(1)  |  DC(2)  ||
  | rwsize = 13 |  DC(3)  |  DC(4)  ||
  | rwsize = 14 |  DC(5)  |  DC(6)  ||
  | rwsize = 15 |  DC(7)  |  DC(8)  ||
ENDTABLE
```

END PVS

As can be seen, DC is an arbitrary "don't care" function, and is given the same type as the output type of the requirements table. Our initial approach was to use an axiom to instantiate DC to any value, but this could be used to introduce a contradiction. Instead, the correctness theorem was modified to account for only those cells in the table which represent valid data. In PVS, predicate sub-typing may also be used for this purpose.

6.3.2 Application to Shakti and the Translation Process

Before translating the Shakti source code into PVS using BAPIP, it was first necessary to determine which sub-component the above derived specification applied to. We discovered that Shakti's implementation of the Byte Lane decoding occurs in the `RapidIO_TgtDecoder_ByteCnt_ByteEn.bsv`. This is a very low-level package in the Shakti Logical Transport subsystem hierarchy, and contains no module. Rather, this functionality is encapsulated by a BSV function, the hardware interpretation of which is a combinational circuit which holds no memory elements.

Once the correct file to translate was identified, the BAPIP translation algorithm was applied in `BSV2PVS` mode. The average run-time was 0.7274s.

6.3.3 Generating a Proof of Correctness

The first step to generating the proof of correctness is formulation of the correctness theorem. Since the functionality we are verifying is encapsulated by a function rather than a BSV clock cycle, it is unnecessary to invoke state transitions. This would be impossible anyways, as a package with no module can hold no state, and therefore can generate no state transitions. The correctness theorem used is listed below:

PVS

```
correctness : theorem
 FORALL (wr_read:bool, wr_Size:Size, wd_ptr:Bit(1)) :
     (((wr_Size <= 10) AND (wd_ptr = 0 OR wd_ptr = 1))
  OR ((wr_Size = 11) AND (wd_ptr = 0)))
   implies (fn_ByteCountDecoder( wr_read
                               , wr_Size
                               , wd_ptr
                               )`byteen_dec =
     req_ByteEnable(wd_ptr, wr_Size))
```

END PVS

In natural language, this might be expressed as “For all possible inputs, if the table entry is valid, the `byteen_dec` field of the result of the function `fn_ByteCountDecoder` matches the result of the requirements specification `req_ByteEnable` for all the same inputs.” Table validity being defined by `wr_Size` being less than 11, or the case where `wr_Size` is 11 and `wd_ptr` is 0. Syntactically, the preconditions imply an equality between the function invocation (with the appropriate field of the structure selected) and the result of the requirements table. It is important to note that the same variables are used as arguments for both sides, to ensure equality of the input parameters of the function and table invocations.

6.3.4 *Proving the Sequent*

In order to prove the above sequent, the following steps were taken.

- `(skolem!)` $\rightarrow$ Skolemization of universally quantified variables.
- Repeated applications of `(flatten)` and `(split)` $\rightarrow$ Organizes the proof and splits it into sub-proofs for the various values and ranges of `wd_ptr` and `wr_Size`.
- `(smash)` $\rightarrow$ Each sub-proof could then be addressed by the general purpose proof strategy `smash`.

In this example, `grind` was not used because it caused an error in the underlying Steel Bank Common Lisp (SBCL) (Steel Bank Common Lisp) installation. Previously, this proof was entirely dischargable using `grind`, with no other human input required. As given in the PVS prover guide (), the `smash` strategy consists of repeated applications of binary decision diagram simplification, if-lifting, and the PVS `assert` strategy, which attempts further simplification and attempts to complete the sub-proof.

Discharging all proof obligations takes an average of 0.17s on the author's computer (ASUS TUF506QM, AMD Ryzen 7 5800H, 16GB RAM, running Linux Mint 20.3). All files used in this case study are available in §F.2, or in the BAPIP project github repository: `https://github.com/nmoore771/bapip`. The lower-than-average proof time of this case study may be due to the fact it contains no state, or it may be the difference in strategies previously mentioned.

6.3.5 Limitations of this Case Study

The obvious limitation of this case study is that the above example does not make use of the action arbitration system detailed in previously in §4.

6.4 RapidIO Read Size and Word Pointer Encoder Module

The case study presented in this section demonstrates several key operations of BAPIP, including the ability to translate and verify modules with wire interactions. In many ways, this proof is symmetrical to that presented in §6.3. From a high level, §6.3 deals with the extraction of pertinent details related to memory transactions from packets, which must occur as specified in the RapidIO specification (RapidIO.org, 2017). Whereas §6.3 deals with packet decoding, this section deals with packet encoding, or the reconstruction of values which were decoded by the previous module. This has been implemented as a stateful Bluespec module with rules which require the action arbitration algorithm.

6.4.1 Objective

We wish to verify the output values of the Shakti RapidIO implementation module `RapidIO_InitEncoder_WdPtr_Size.bsv`. Specifically, we wish to verify the behaviour of `reg_Size` and `reg_WdPointer`. These values are accessible using output methods `outputs_Size_` and `outputs_WdPointer_` respectively. These outputs are derived from the inputs to the methods `_inputs_Read`, `_inputs_ByteCount`, and `_inputs_ByteEn`. The relationship between these inputs and outputs is given in the RapidIO Specification RapidIO.org (2017).

6.4.2 Formalization of the RISC-V Specification

From a high-level perspective, the specification we are using for this case study is the same one used in §6.4.4, read backwards. This has mainly to do with the module under examination reconstructing the data derived from the incoming packet which the module in §6.3 decodes. The outputs of one are the inputs of the other, and vice versa.

6.4.3 Performing the Translation

The invocation of BAPIP to generate the PVS encoding of the module under examination proceeded according to the structure outlined in §A.4. However, the selections made during the selection of the top-level method invocations should be examined.

`RapidIO_InitEncoder_WdPtr_Size.bsv` has five methods, two of which are output methods. We are interested in verifying both of them, and both are dependent on values provided by all three of the input methods. It is therefore necessary to generate a transition predicate using all three of them. Further, the calculation at hand takes a single clock cycle to execute, so the above mentioned transition is also the only one required. That is, we have no need to chain multiple transition predicates together.

6.4.4 Formalization of the RISC-V Specification

As in the previous case, the use of the table in Figure 6.3 hinges on the first two columns of each row presenting a unique combination from which the values in the second two columns may be unambiguously extracted. However, there is one major difference, mathematically speaking, between the interpretation presented here and that in §6.3. The combination of word pointer and read size is complete and disjoint, with respect to determining values for both the number of bytes and the byte lanes. There are 32 table entries and five bits of complexity between the two, so if each table entry is unique (true upon examination), this also means that all possible values are represented. However, in the reverse case, we have 16 bits of complexity at play. While each combination of the number of bytes and the byte-lanes is unique, it is wholly impossible to have every value represented, as our table would need

Number of Bytes	Byte Lanes	wdptr	rdsize
1	0b10000000	0b0	0b0000
1	0b01000000	0b0	0b0001
1	0b00100000	0b0	0b0010
1	0b00010000	0b0	0b0011
1	0b00001000	0b1	0b0000
1	0b00000100	0b1	0b0001
1	0b00000010	0b1	0b0010
1	0b00000001	0b1	0b0011
2	0b11000000	0b0	0b0100
3	0b11100000	0b0	0b0101
2	0b00110000	0b0	0b0110
5	0b11111000	0b0	0b0111
2	0b00001100	0b1	0b0100
3	0b00000111	0b1	0b0101
2	0b00000011	0b1	0b0110
5	0b00011111	0b1	0b0111
4	0b11110000	0b0	0b1000
4	0b00001111	0b1	0b1000
6	0b11111100	0b0	0b1001
6	0b00111111	0b1	0b1001
7	0b11111110	0b0	0b1010
7	0b01111111	0b1	0b1010
8	0b11111111	0b0	0b1011
16		0b1	0b1011
32		0b0	0b1100
64		0b1	0b1100
96		0b0	0b1101
128		0b1	0b1101
160		0b0	0b1110
192		0b1	0b1110
224		0b0	0b1111
256		0b1	0b1111

Tab. 6.3: Table 4-3 from (RapidIO.org, 2017) slightly rearranged

to be very long indeed. Therefore, the table as interpreted for this case study is disjoint, but not complete. As such, the manner of encoding these conditions in PVS has changed somewhat from §6.3, as will be demonstrated in a code snippet included in §6.4.4. This situation is compounded by the fact that neither the byte lanes nor the number of bytes is complete individually either.

It is also worth noting that the last nine entries in the table are uniquely determined by the number of bytes alone, and that the byte lanes are not even specified. This is likely because byte-lanes are only useful when the data being accessed is smaller than a word (64 bits in this case), and that the number of bytes for these last entries are multiples of 8, indicating whole word operations.

The obvious question that arises from this lack of completeness is how to interpret cases not represented by the table. There are two approaches that are illustrated in this case study.

Assumed Default Values

Firstly, one may introduce an assumption about behaviour in the implicit cases. Normally in such situations, it is reasonable to assume that some default value would be referred to, and zero is a common default value. For this table, the requirements have been expressed in PVS as nested `COND` blocks. A `COND` block in PVS is syntactic sugar for a set of if-then-else statements, where the final entry in the COND block is presumed as the `else` case, in addition to generating type correctness conditions for disjointness and completeness. So, if we wish to encode an incomplete table as a `COND` block, attention must be paid to the terminal expression. In a naive implementation which omits an explicit `else` case, the terminal entry in the `COND` table will be assumed to be the `else` case, as conditionals in PVS may not be incomplete. Although these implicit cases are not addressed in the table, it is unlikely that this is a valid interpretation. Therefore, the expressions below have explicit `COND` block entries which yield an assumed default value of zero.

PVS

```
req_word_pointer( bytemask : Bit(8)
                , bytecount : Bit(8)
                ) : Bit(1)
 = COND
    bytecount = 1   -> COND
        bytemask = 8  -> 1
      , bytemask = 4  -> 1
```

```
        , bytemask = 2   -> 1
        , bytemask = 1   -> 1
        , bytemask = 128 -> 0
        , bytemask = 64  -> 0
        , bytemask = 32  -> 0
        , bytemask = 16  -> 0
        ENDCOND
    , bytecount = 2   -> COND
          bytemask = 12 -> 1
        , bytemask = 3   -> 1
        , bytemask = 192 -> 0
        , bytemask = 48  -> 0
        ENDCOND
    , bytecount = 3   -> COND
          bytemask = 7   -> 1
        , bytemask = 224 -> 0
        ENDCOND
    , bytecount = 4   -> COND
          bytemask = 15 -> 1
        , bytemask = 240 -> 0
        ENDCOND
    , bytecount = 5   -> COND
          bytemask = 31 -> 1
        , bytemask = 248 -> 0
        ENDCOND
    , bytecount = 6   -> COND
          bytemask = 63 -> 1
        , bytemask = 252 -> 0
        ENDCOND
    , bytecount = 7   -> COND
          bytemask = 127 -> 1
        , bytemask = 254 -> 0
        ENDCOND
    , bytecount = 8   -> COND
          bytemask = 255 -> 0
        ENDCOND
    , bytecount = 16  -> 1
    , bytecount = 32  -> 0
    , bytecount = 64  -> 1
    , bytecount = 96  -> 0
    , bytecount = 128 -> 1
    , bytecount = 160 -> 0
    , bytecount = 192 -> 1
    , bytecount = 256 -> 1
    , bytecount = 224 -> 0
    , True -> 0
      ENDCOND

req_read_size ( bytemask : Bit(8)
              , bytecount : Bit(8)
              ) : Bit(1)
 = COND
      bytecount = 1   -> COND
          bytemask = 128 -> 0
        , bytemask = 64 -> 1
        , bytemask = 32 -> 2
        , bytemask = 16 -> 3
        , bytemask = 8  -> 0
        , bytemask = 4  -> 1
        , bytemask = 2  -> 2
        , bytemask = 1  -> 3
        , True -> 0
        ENDCOND
    , bytecount = 2   -> COND
          bytemask = 192 -> 4
        , bytemask = 48 -> 6
        , bytemask = 12 -> 4
        , bytemask = 3  -> 6
        , True -> 0
        ENDCOND
    , bytecount = 3   -> COND
          bytemask = 224 -> 5
        , bytemask = 7  -> 5
        , True -> 0
        ENDCOND
    , bytecount = 4   -> COND
          bytemask = 240 -> 8
        , bytemask = 15 -> 8
        , True -> 0
        ENDCOND
```

```
, bytecount = 5   –> COND
      bytemask = 248 –> 7
    , bytemask = 31 –> 7
    , True –> 0
    ENDCOND
, bytecount = 6   –> COND
      bytemask = 252 –> 9
    , bytemask = 63 –> 9
    , True –> 0
    ENDCOND
, bytecount = 7   –> COND
      bytemask = 254 –> 10
    , bytemask = 127 –> 10
    , True –> 0
    ENDCOND
, bytecount = 8   –> COND
      bytemask = 255 –> 11
    , True –> 0
    ENDCOND
, bytecount = 16  –> 11
, bytecount = 32  –> 12
, bytecount = 64  –> 12
, bytecount = 96  –> 13
, bytecount = 128 –> 13
, bytecount = 160 –> 14
, bytecount = 192 –> 14
, bytecount = 224 –> 15
, bytecount = 256 –> 15
, True –> 0
ENDCOND
```

END PVS

One clear and obvious drawback of nested `COND` blocks vs `TABLE` notation is that it is much less compact. Tables could have been used here with equal effect, but `COND` blocks were used to minimize the number of explicit "don't care" values needed, since these tables had a reasonably large number of them.

Restriction of input range

There is a second argument that can be made in order to solve the incompleteness problem. One can take the position that in the consideration of cases not explicitly addressed by the specification, the module under examination may behave however it likes, so long as it is compliant with those conditions explicitly stated. This essentially puts an implicit “Don’t Care” value in any case not explicitly addressed.

The way we might interpret this in PVS is that we are only concerned with the requirements relation holding for the specific input values indicated by the table. One way to introduce this condition to our eventual theorem is to create a predicate which is only true when given a “valid” value for the input under examination. Valid here means a value addressed in the table explicitly. The following PVS encodes these predicates.

PVS

```
valid_bytemask (bytemask : Bit(8)) : bool =
     bytemask = 128
  or bytemask = 64
  or bytemask = 32
  or bytemask = 16
  or bytemask = 8
  or bytemask = 4
  or bytemask = 2
  or bytemask = 1
  or bytemask = 192
  or bytemask = 48
  or bytemask = 12
  or bytemask = 3
  or bytemask = 224
  or bytemask = 7
  or bytemask = 240
  or bytemask = 15
  or bytemask = 248
  or bytemask = 31
  or bytemask = 252
  or bytemask = 63
  or bytemask = 252
  or bytemask = 63
  or bytemask = 254
  or bytemask = 127
  or bytemask = 255

valid_bytecount (bytecount : Bit(8))  :  bool =
     bytecount = 1
  or bytecount = 2
  or bytecount = 3
  or bytecount = 4
  or bytecount = 5
  or bytecount = 6
  or bytecount = 7
  or bytecount = 8
  or bytecount = 16
  or bytecount = 32
  or bytecount = 64
  or bytecount = 96
  or bytecount = 128
  or bytecount = 160
  or bytecount = 192
  or bytecount = 224
  or bytecount = 256
```

END PVS

These predicates not only restrict the search space of the deduction in a very useful way, but also provide an easy way to decompose the proof into sub-proofs that are more digestible to the automatic proof strategies PVS incorporates.

Combining the approaches

Our approach to this case study uses both of the above methods simultaneously. While it is true that taking the input restricting predicates as antecedents means the `else` cases are hypothetically never exercised, it is the position of the author that the inclusion of the `else` cases make the requirement expressions themselves more accurate with respect to the intention of the specification document, so they are kept. Further, the predicates restricting input values create such a powerful savings in terms of proof execution time that it is prudent to keep them, regardless as to their strict necessity.

It should be noted that the input restrictions specified in §6.4.4 specify valid values for each input separately. The search space may be considered the product of these two search spaces, and there are still a large number of combinations which will be examined by PVS that are not valid table entries.

6.4.5 Derivation of a Theorem

Now we have all the pieces in place to be able to construct our sequents. The approach taken in the three theorems below is to separate our concerns with respect to demonstrating the correctness of the word pointer and read size outputs. Both theorem 1 and 2 have identical antecedents, and we could therefore compose them into a single sequent. This would be functionally equivalent, but it is the contention of the author that the approach given enhances readability. The third theorem is simply the composition of the two correctness theorems.

PVS

```
correctness_1 : theorem
 forall(x1 : ByteEn, x2 : ByteCount, x3 : bool) :
      x3 = True
  and valid_bytemask(x1)
  and valid_bytecount(x2)
  and transition(1, s(0),s(1), x1, x2, x3)
  implies req_word_pointer(x1,x2)
    = outputs_WdPointer_(1,s(1),s(1),x1,x2,x3)

correctness_2 : theorem
 forall(x1 : ByteEn, x2 : ByteCount, x3 : bool) :
      x3 = True
  and valid_bytemask(x1)
  and valid_bytecount(x2)
  and transition(1, s(0),s(1), x1, x2, x3)
  implies req_read_size(x1, x2)
    = outputs_Size_(1,s(1),s(1),x1,x2,x3)

  correctness_total : theorem
    correctness_1 and correctness_2
```

END PVS

We start by universally quantifying over the three input values. The third, `x3`, corresponds to `wr_Read` within the module. This wire seems to have some internal purpose, and does not appear in the specification. Examination of the code reveals that read mode must be indicated in order for the behaviour of this module to be in accordance with the specification. The functionality of the module appears to be compounded with either a write or hybrid mode. For the purposes of these proofs, we set the module in read mode by including as an antecedent that `x3` is `True`. We then add as antecedents both our input validity predicates and our transition predicate. Since this module arrives at its output in one clock cycle, only one transition predicate is necessary. As consequent, we call the appropriate output method and test its return value against the output of the requirements table.

6.4.6 Proving The Sequent

The organization of this case study is somewhat misleading as to the process of proof development. While it is necessary to derive a provisional theorem as a base to build the proof upon, in truth the sequent must often undergo a process of refinement in order to render the proof achievable via some combination of manual strategies and automatic deduction. This process is reflected above, but the reader should bear in mind that failed proof attempts were used in this case to refine the theorem. In this manner, the theorem and proof may correctly be considered to have been co-developed.

It must be noted before we begin that the general methodology for all the proofs presented in this thesis is to check first whether a sequent may be proven completely automatically. The failure of automatic deduction is not normally the production of a counter example, especially in proofs which will in any event result in successful deduction. Rather, this is normally a matter of impatience on the part of the proof author. It is the general observation of this author that if a proof is not completed within two or three minutes, there is some possibility of it being completed within 15. However, if a proof has taken 15 minutes and is not completed, the probability of the proof being completed within an hour is small. At this point, application of some introductory strategies to break the proof down into more automatically digestible components results in faster total run-time than allowing the proof system to persist in its automatic strategies, even if some degree of human supervision is necessary.

The latter approach was necessary in the case of the above sequent. We will now detail the methodology used to break the proof down and solve it.

1. At the top-level, our overall correctness theorem is the conjunction of the correctness theorems addressing for the word pointer and read size registers (`correctness_1` and `correctness_2` respectively). The proof is immediately divided into two sub-proofs for these two sub-theorems using `(split)`. There is no difference whatever between the sequence of strategies addressing these two branches, the following steps were applied to both. For the purposes of this description, we will follow the process for `correctness_1`.
2. The theorem is expanded using `(expand correctness_1)`
3. Skolemization is performed over the universal quantification, by using `(skolem!)`.
4. The top level implication and conjunctions are applied as antecedents and consequents using `(flatten)`.
5. The definition of the `valid_bytemask` predicate is expanded using `(expand valid_bytemask)`
6. The proof is then split into 25 sub-proofs, along the disjunctions of the newly expanded predicate using `(split)`.
7. Each of the resulting 25 sub-proofs is then digestible to the general-purpose automatic proof strategy `(grind)`. Application of this strategy to each of the 25 sub-proofs results in the complete discharging of all proof obligations.

Using the above strategy, the proof of `correctness_total` is discharged with an average execution time of 18.28s on the author's computer (ASUS TUF506QM, AMD Ryzen 7 5800H, 16GB RAM, running Linux Mint 20.3). All files used in this case study are available in §G.2, or in the BAPIP project github repository: `https://github.com/nmoore771/bapip`.

6.4.7 In Conclusion

As demonstrated via the above process of formal logical deduction, we can state with confidence that `RapidIO_InitEncoder_WdPtr_Size.bsv` is in compliance with revision 4.1 of the RapidIO Interconnect Specification (RapidIO.org, 2017), insofar as the byte size and word pointer fields of outgoing packets are correctly constituted.

6.5 Progress Towards RapidIO Transaction ID Echoing

The goal of the BAPIP project was to produce a series of proofs of concept for the translation/proof workflow, over a sequence of hardware modules of increasing complexity. The transaction ID echoing discussed in this case study should be viewed as a concept which the author was ultimately unable to prove, for a variety of reasons which we will explore.

In this case study we shall explore the property to which the BAPIP tool and process were applied, explore how that property might be more formally stated, describe how the process could be applied to it, and finally, discuss what went wrong and the limitations to the methodology this demonstrates.

6.5.1 The Problem Attempted

The RapidIO specification describes the interpretation of data packets transmitted via a RapidIO compliant interconnect fabric. The first four bits of any data packet indicate packet type, for example, `READ`, `WRITE`, `MAINTENANCE`, etc. One of the fields encoded in most packet types is a transaction ID. Transaction IDs are generated by a requesting element, and are used to distinguish packets. In RapidIO, it is possible for many separate, multi-packet interactions to be occurring simultaneously on the same network. Transaction IDs can even be used to determine sequencing of transactions, presuming all elements are synchronizing their transaction ID generation.

The goal of this case study was to verify the transaction ID data pathway. The package hierarchy of the RapidIO logical transport system is given in Figure 6.7.

Proving this data pathway requires the embedding of most of the package hierarchy listed in Figure 6.7 in PVS. For context, the `MainCore` package alone contains over 100 methods.

6.5.2 Derivation of a Formal Property

RapidIO is fundamentally a message passing system. This means that many different types of components are intended to be able to communicate with

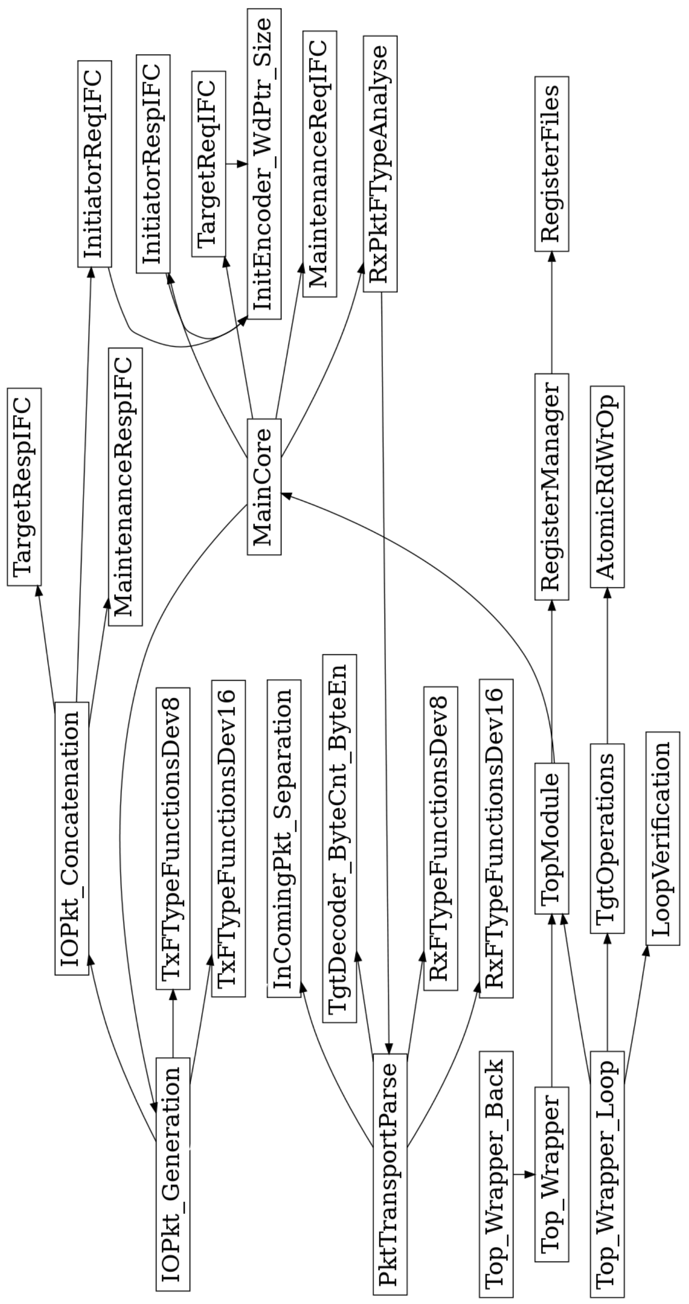


Figure 6.7: Shakti RISC-V Module Hierarchy Diagram

each other via RapidIO. It makes sense, therefore, to abstract away the internal mechanisms of the component as far as possible, and focus on the interface itself. We can decompose our desired property, “RapidIO must handle transaction IDs in accordance with the specification,” into “given correct behaviour in the parent component, RapidIO must interpret an incoming packet correctly with respect to the transaction ID, and RapidIO must generate packets with the transaction ID in the correct location.”

Fortunately the RapidIO specification gives specific locations within the various packet formats where the transaction ID is to be located. These are tabulated below. Packet types with no fields specified (those unaddressed in Chapters 1 and 2 of (RapidIO.org, 2017)) have transaction ID locations labelled as “not applicable”. For packet types that are specified, but which do not contain a transaction ID (types 6 and 11), “none” has been used. Please note also that we are also only dealing with the “Logical Transport” version of the RapidIO interconnect (RapidIO.org, 2017), and as such, packet format specifications are taken strictly from Chapters 1 and 2 of the RapidIO specification.

It is also necessary to know how many clock cycles the specified behaviour will take. This must be reflected directly in the number of transitions invoked in the theorem. The only ways to determine this are foreknowledge of the system, observation of non-formal simulation, or observation of the source code. In our case, we shall proceed by observing the source code.

Figures 6.8 and 6.9 depict the flow of data through the module hierarchy in the case of received and transmitted packets respectively. Source code for these data pathways may be found in §H.1. The order of memory transactions (indicated by arrows) is expressed using Arabic numerals. The numbers to the left of the memory elements indicate the number of clock cycles that have elapsed since the first during the access of the given memory element. While this gives an overall picture of the flow of data, and the number of clock cycles required to generate a response packet, the data flow depicted is one of a number of possible paths. In the interests of simplifying an already complex diagram, one representative data pathway will be displayed. The only operations which increase the number of clock cycles used are register writes and FIFO enqueue operations.

In the case of Figure 6.8, parsing of the data packet, and the arrival of the transaction ID in its final register, takes 14 steps over a course of 3 clock cycles. Steps include method invocations, wire write, register write and FIFO operations. The data passes through five BSV modules before finally being

ftype	Packet Type	Packet Description	TID bits
0000	Request	implementation dependant	n/a
0001	Request	Reserved	n/a
0010	Request	ATOMIC / NREAD	[12:19]
0011	Request	Reserved	n/a
0100	Request	Reserved	n/a
0101	Request	ATOMIC / NWRITE	[12:19]
0110	Request	SWRITE	none
0111	Request	Reserved	n/a
1000	Request	MAINTENANCE	[12:19]
1001	Request	Reserved	n/a
1010	Request	DOORBELL	[12:19]
1011	Request	MESSAGE	none
1100	Response	Reserved	n/a
1101	Response	RESPONSE	[12:19]
1110	Response	Reserved	n/a
1111	Response	implementation dependant	n/a

Tab. 6.4: Packet Types and Transaction ID Bit Ranges

stored in `MainCore`. The complexity of this datapath was the primary reason this case study did not reach a successful conclusion.

In the case of Figure 6.9, packet construction is not as complex as packet decomposition. We pass through 13 steps over the course of 2 clock cycles. While the number of steps overall is lower than with parsing, the number of steps per clock cycle is higher. This data path passes through four hardware modules.

From a top level perspective, verification of the transaction echoing property can be expressed in two parts. For parsing, given the register in which a data packet is received, and knowing which register the transaction ID should be parsed to, can we demonstrate that what goes in also comes out? Similarly for packet generation. Knowing the two endpoints of the data pathway, and knowing that this pathway should not modify the transaction ID, do we get out what we put in?

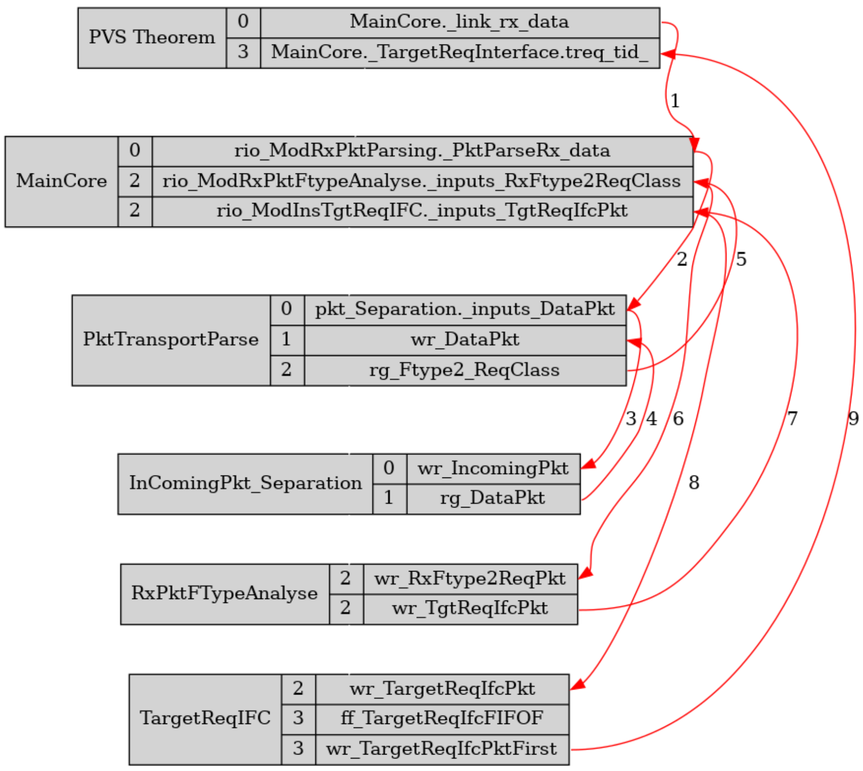


Figure 6.8: Flow of Data Through Parsing

6.5.3 Translator Configuration

In order to begin formulating the above property as a formal theorem in PVS, the translator must generate a transition schedule that encapsulates the desired behaviour. While there are a large number of methods contained in `RapidIO_MainCore.bsv`, as seen in §H.1.4, the overwhelming majority of them are irrelevant. Specifically, they provide a means of accessing many intermediate values throughout the subsystem. We are interested primarily in two methods: `_link_rx_data` and `link_tx_data_`, `rx` and `tx` being hardware shorthand for receiving and transmitting respectively. Please note how `_link_rx_data` takes a value of type `DataPkt` as an argument, and how `link_tx_data_` returns a value of the same type. Note also that these two functions are the only functions to operate on this data type, which is used by the Shakti RapidIO implementation to represent the data packets discussed in the section above.

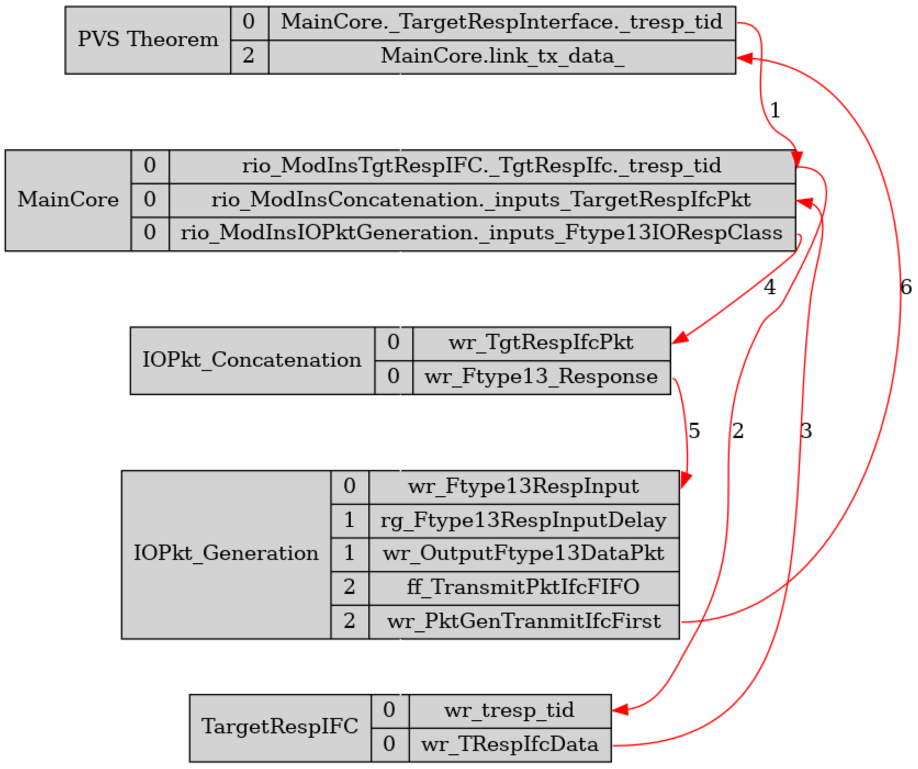

PVS Theorem
0 MainCore._TargetRespInterface._tresp_tid
2 MainCore.link_tx_data_
MainCore
0 rio_ModInsTgtRespIFC._TgtRespIfc._tresp_tid
0 rio_ModInsConcatenation._inputs_TargetRespIfcPkt
0 rio_ModInsIOPktGeneration._inputs_Ftype13IORespClass
IOPkt_Concatenation
0 wr_TgtRespIfcPkt
0 wr_Ftype13_Response
IOPkt_Generation
0 wr_Ftype13RespInput
1 rg_Ftype13RespInputDelay
1 wr_OutputFtype13DataPkt
2 ff_TransmitPktIfcFIFO
2 wr_PktGenTranmitIfcFirst
TargetRespIFC
0 wr_tresp_tid
0 wr_TRespIfcData
1
2
3
4
5
6

Figure 6.9: Flow of Data Through Concatenation

While `link_tx_data_` is accessible at any time, due to it being a `Value` method containing no state implication, `link_rx_data_` does indeed have state implication, somewhat obviously. As such, any schedule generated must include `link_rx_data_`. As a precautionary measure, one might include all methods within the `Ifc_LinkInterfaceRx` interface in the generated schedule. The mechanism for specifying specific methods to be included as top-level calls for clock cycles under consideration during scheduling is discussed in §A.4. Once the appropriate methods were specified, the following command was used.

```
$ stack exec BAPIP-exe bsv2pvs <dir>/RapidIO_MainCore.bsv
    RapidIO_MainCore <dir>/RapidIO_MainCore
```

6.5.4 Current State of the Translation

At time of writing, the proximal cause of the failure of this case study is that running the translation algorithm over the entire RapidIO subsystem results in code which does not pass PVS's typechecker. Specifically, the error occurs during the translation of value methods. Within the top-level file in the RapidIO library, a couple of output methods exercise a corner case which currently fails to translate correctly. The problem is caused by the intersection of three factors.

- The value which is being returned is of the `DWire` type.
- This wire is a structure type
- This wire's structure type is encapsulated by a Maybe type.

Variables of the `DWire` are introduced as locally bound variables ("let" statements), where the state specific tree specific to said wire is the value of this locally bound variable. During let binding generation, a data type must be attributed to the locally bound variable. For some unknown reason, the translator is failing to encapsulate the structure type in the Maybe type, which is causing a type inconsistency with the return type of the value method itself. Something specific about the combination of these factors causes the described error. In other portions of the program, each of these factors typecheck by themselves.

It should be noted that it is unlikely that this is the only outstanding error. In the opinion of the author, at least four months of development time would be needed to work out remaining errors, and it could be more than that, since the number of remaining errors is unknown. This extended development time is due primarily to the complexity of the translation program and the task being attempted, and is in line with the amount of time other work on the translator has taken.

So What Went Wrong?

The considered opinion of the author is that the goal of verifying the entirety of the Shakti RapidIO library was not sufficiently scaffolded by the demonstration of case studies of intermediate difficulty. The goal of verifying the entire library from the top-level was a lofty one, spurred on by the desire to have achieved a highly impressive result.

A cautious and gradual approach may have resulted in greater success in this respect. If with each new file in the library, the time was taken to develop a case study which explored the new features which had to be added to accommodate said file, it would have been far easier to isolate and eliminate the errors. In the opinion of the author, an additional six or eight case studies would provide have provided a much better scaffold for the top-level module verification. Further, a longer series of case studies of increasing complexity would have provided more assurance as to the validity of the translation and verification methodology. Instead, we tried to rush to the end goal of the project, missing several rungs on the ladder, and failed to reach the stated goal.

Of course, the production of additional case studies would have taken more development time, and it is unlikely that the entirety of the library would have been attempted. Such a thing would require additional time and personnel resources to execute successfully.

It is also possible that the abstraction mechanisms built into the translator could be improved, and thus the debuggability of the software improved. A better system of abstractions would, however, require reimplementing the core BAPIP algorithm, and re-proving everything under the new system. The core BAPIP algorithm has been similarly rebuilt four times over the ten year lifespan of the project, and always for this reason. While a fifth time is hypothetically possible, and such an attempt may have scientific value. The algorithm at this point has been pushed far beyond the scope of the original work by Richards and Lester (Richards and Lester, 2011), and succeeds in verifying whole clock cycle semantics (see §6.4). While the need for a greater degree of abstraction is recognized, the exact nature of that abstraction is unknown. Throughout the remodellings of BAPIP over the years, the core abstraction of Kripke semantics has been maintained. It is suspected that the limits of that model have been reached, and whatever comes next would have to use some other formalism less susceptible to the state explosion problem.

7. CONCLUSION

Through application, the utility of the method of formal mathematical verification discussed herein has been demonstrated. Several case studies have been presented applying our multi-purpose translation software BAPIP to a number of verification problems of varying complexity. The methodology, construction of the tool, and manner of its operation have all been discussed in detail.

It is our hope that this technique may be used and adapted in further verification projects, so that the immense barrier between formal verification and common practice may be lessened, resulting in more widespread adoption of mathematically rigorous technique, and the elevation of the discipline of Software Engineering to the same standards of reliability and repeatability enjoyed by other Engineering disciplines.

7.1 Summary of Empirical Data

The following is a summary of empirical data related to the translation software developed, and the case studies presented.

7.1.1 Translation Software

As a rough approximation, the BAPIP translator itself took around 70 person-months to develop. Table §7.1 gives statistics on the translator source code. Table §7.2 gives statistics of interest on all translations of BSV source code discussed in this thesis.

The following is a summary of the assurances gained through this study.

- The implementation of the `LIMITS_ALARM` function block presented is compliant with IEC61131-3 (IEC, 2013)

File Name	**LoC**	**LoC (no Com.)**	**Functions**
`BAPIP.hs`	381	325	23
`BSV2PVS.hs`	4095	3738	409
`BSVGenerator.hs`	401	374	66
`BSVLexer.hs`	2084	1935	93
`ConflictSolver.hs`	539	431	44
`HEXLexer.hs`	102	90	10
`LexerTypes.hs`	590	443	7
`LiteralLexer.hs`	239	208	29
`MacroProcessor.hs`	270	212	29
`PVS2BSV.hs`	26	21	4
`PVSGenerator.hs`	2093	1830	236
`SourceFiles.hs`	32	30	10
`TSP2BSV.hs`	512	438	69
`TSPLexer.hs`	756	667	45
Total	**12,070**	**10,698**	**1070**

Tab. 7.1: BAPIP Source Code Statistics

- The tabular specification for the `ALRM_INT` function block, found in the IEC61131-3 (IEC, 2013) was used to generate a BSV hardware description, and that description was demonstrated compliant with the same.
- The Shakti implementation of RapidIO is compliant with the RapidIO specification (RapidIO.org, 2017) in the following areas:
  - When decoding an incoming packet, the positions of enabled bytes is compliant.
  - When encoding the same, the packet fields are filled out correctly, based on the number and position of bytes requested.

7.2 *Applicability of Work*

While the work presented herein has expanded our ability to automatically verify BSV hardware descriptions, it is important to discuss the limitations

Section Reference	§4.2.1	§6.1	§6.2	§6.3	§6.4	§6.5
Input BSV Files	2	3	1	4	3	24
Input BSV Lines	86	61	41	1136	1192	8171
Input Characters	1898	1238	755	43934	42952	361208
Output PVS Files	11	11	12	8	11	11
Output PVS Lines	771	801	873	1336	2297	6582
Output Characters	20128	21490	22200	37687	56477	247078
Translation Time	0.234s	0.203s	0.211s	0.547s	0.542s	19.918s
Proof Time	n/a	0.66s	0.52s	0.17s	18.28s	n/a

Tab. 7.2: BAPIP Case Study Statistics

of the applicability of this methodology. One such limitation is that the module we wish to prove must be implemented in Bluespec SystemVerilog. Fortunately, Logical Equivalence Checkers (LECs) can address this deficiency. LECs analyze two hardware descriptions for black-box logical equivalence. They are off-the-shelf software tools, and have been successfully used to check the Hysteresis block presented in our case studies (which had been verified in PVS) against an independent implementation of the Hysteresis block in VHDL. By so doing, our proof of correctness of the BSV block is transferable to the VHDL implementation, in so far as the Bluespec compiler is correct.

The automated model extraction performed by BAPIP is not total over Bluespec SystemVerilog. It is our position that the subset of Bluespec SystemVerilog for which our software operates expresses meaningful and industrially practical hardware descriptions, despite not accepting the entire language. It is also important to note that the translation is not injective into PVS, so a future reverse translation for the purposes of round-trip engineering would not be able to exactly reconstruct the original BSV file. Any reverse translation must necessarily operate on only the set of PVS files that are generatable by the translator, which is a fairly restrictive subset despite the wide scope of translatable BSV. The object would not be to produce a general translation from PVS to BSV, but only to enable the reversion of slightly modified translator output files back into their original language. Given the complexity of the PVS output, this functionality may never pass into the realm of practicality for designs of non-trivial size.

It is also important to note that Bluespec cannot duplicate all functionality expressible in lower-level hardware description languages, due to the hardware elements added by the Bluespec compiler to create and control properties like rule scheduling and atomicity. It is therefore possible that a hardware description in a lower-level language could conform to any given requirements and not be expressible in BSV.

7.3 Alternative Approaches

During the development of any large project like BAPIP, key decisions made early in the project have a pronounced effect on project outcomes. Some of these key decisions will now be explored.

Most of the very early key decisions in the development of BAPIP were made by adopting the framework proposed by Richards and Lester (2011). This set the source and target language of the translation, as well as mathematical model of computation.

With respect to the source language BSV, alternative high-abstraction hardware languages are scarce, and it is the author's opinion that BSV has been a serviceable language to attempt to verify. By the time the project had entered its later stages, it was clear that Bluespec did not quite live up to the claim of having an "elegant semantic" making it particularly amenable to translation. All the formal methods projects operating on BSV, except this one, fall short of modelling the action arbitration mechanism, and in particular the scheduling complications introduced by wires. Exclusion of wires *severely* restricts expressible hardware designs; it is a feature that a hardware designer would expect to have. Despite this, Bluespec is still preferred to the majority market-share HDLs, like VHDL and SystemVerilog. The ad-hoc nature of these languages increases the effort needed for verification quite drastically, particularly for a static analysis engine like BAPIP.

With respect to the target language PVS, there were a number of alternatives. Among the projects mentioned in §1.4.1, Coq was a particular favourite, but there are a large number of other theorem provers that would have served our purpose. The choice to use PVS was again primarily made in following the work of Richards and Lester. If this project had been attempted after some of these other projects had been published, projects such as Kami or Fe-Si could have been used in theory as a basis for an action arbitration extension. The author reserves judgment on this, however, as this was the planned course of action with the Richards and Lester translation, and almost the whole of it had to be thrown out in order to accomplish what has been accomplished. If the project were to be rebuilt entirely, the author would investigate the possibility of embedding the property-proving aspect of the project in the translator program itself. If some suitable form of input could be found for the original specifications, it might be possible to accomplish all that PVS currently accomplishes using the SBV library and Yices. Alternatively, other theorem provers such as HOL4, Isabelle or Event B would be considered.

7.4 Updates Regarding Recent Software Releases

7.4.1 Open Source Bluespec Compiler Release

During the majority of this project, BSC (the Bluespec Compiler) was a closed-source tool. The creation of BAPIP required the reverse-engineering of many aspects of the compiler and language features. Since the open-source release of the compiler (Bluespec Inc., 2020), some observations can be made as to the direction the project would have taken, had this source code been available earlier.

Two mechanisms of primary interest are the BSC parser and typechecker. In theory, if code could have been adapted from BSC itself, much effort could have been saved in the development of BAPIP. After an examination of the BSC source code, it is evident that much of these crucial subroutines are encoded not in Haskell, as was previously believed, but in C++. Most particularly, the abstract syntax tree generation and typechecking appear to occur in C++. Curiously, the first step in Bluespec compilation appears to be cross-compilation to C. The Haskell method, `parseSrc`, which is used by top level main functions to parse BSV source files, outputs a `CPackage` data type. This data type, when examined, contains a language definition for a sub-language of C. Code comments warn that construction of invalid C programs is possible using said language definition.

The following reasons are hypothesized for why such cross-compilation to C/C++ is necessary.

- It is possible that the compiler needed to be implemented in a faster language.
- It is possible that the compiler needed to interface with some pre-existing compiler (or portion thereof) in order to generate HDL code.

In the opinion of this author, in order to make use of this code directly for the implementation of BAPIP, it would be necessary to completely rewrite the entirety of the translator. The central abstractions of BAPIP are the record data types which encode BSV and PVS abstract syntax, as well as internal representations. These central abstractions are encoded in C++ in BSC, and interfacing these two representations would be highly problematic. The two ASTs, afterall, are ASTs for *different languages*. The BAPIP AST is designed for the Bluespec language, not this previously unknown internal C++ representation.

This internal C++ representation presents one major additional problem: the internal language semantics. While external language semantics have been documented (Bluespec Inc., 2012a), any comparable documentation for the semantic details of the internal language would presumably be internal to Bluespec Inc., and have not been subject to public release, to the best knowledge of the author.

7.5 Future Work

While the work herein presented makes great strides towards to ultimate goal of automatically verified hardware descriptions, there are a number of ways the project could be extended and expanded.

First and most obviously, the bsv2pvs translation algorithm could be extended to encompass the entire language of Bluespec SystemVerilog. This would permit more hardware descriptions to be translated.

As has been alluded to previously, support for a pvs2bsv algorithm has been architecturally included in BAPIP, though the algorithm itself has not been completed. Completion of this algorithm would allow round-trip engineering of BSV files.

There is precedence for this type of system having a specification Domain Specific Language (DSL) (domain specific language) attached, so that the user could specify requirements in some syntactically more convenient manner. This would not only allow for greater ease of use on the part of the end user, but would enable the algorithm to automatically generate both correctness theorems, and even proof strategies for them. In this manner, it is conceivable that BAPIP would return a simple pass/fail when asked to prove a block, which really would be the ultimate in user-friendliness.

Finally, the translation process could be further improved in terms of applicability to large descriptions using a better core abstraction, as well as modelling the following currently unmodelled behaviours:

7.5.1 Other Pragmas

Besides those controlling action arbitration, there are many different pragmas. While many of these are useful for development in Bluespec, the integration of them with the existing translator structure is reserved as future work.

7.5.2 Type Classes and Type Class Declarations

A potentially useful future addition to the translator would be the semantic interpretation of the type classes included in type definitions. These could be used to impose/remove additional constraints from custom types, improving the subset of Bluespec SystemVerilog covered by the translator. Additionally, user-generated type-classes are not addressed.

7.5.3 Parameterized Modules

One of the powerful abstraction mechanisms available in BSV is the ability to create parameterized modules. This capability is not currently supported. Supporting it would require a reasonably drastic re-organization of the post-translation PVS semantics. Currently, the modularity of Bluespec modules is preserved only insofar as they extend to the state record. The implementation of this feature is reserved as future work.

7.6 Summary of Contributions

This thesis has made the following contributions to the state of the art in the verification of BSV hardware designs:

- This work is the first formalization of BSV semantics to include constructs violating one-rule-at-a-time semantics, such as wires and FIFOs.
- This methodology is applicable to descriptions of non-trivial size, drawn from industry, as demonstrated in our case studies.
- BAPIP has made the formal verification of real-world BSV hardware designs less expensive in terms of time and expertise.

BIBLIOGRAPHY

(2020a). containers :: Stackage Server. URL: https://www.stackage.org/lts-14.23/package/containers-0.6.0.1 Accessed March 29, 2022.

(2020b). parse :: Stackage Server. URL: https://www.stackage.org/lts-14.23/package/directory-1.3.3.0 Accessed March 29, 2022.

(2022). mit-plv/koika: A core language for rule-based hardware design. https://github.com/mit-plv/koika.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer. https://link.springer.com/chapter/10.1007/978-3-642-22110-1_14 Accessed March 30, 2022.

Bertot, Y. (2008). A short presentation of coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 12–16. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-540-71067-7_3 Accessed August 11, 2022.

Bidmeshki, M. M. and Makris, Y. (2015). Toward Automatic Proof Generation for Information Flow Policies in Third-Party Hardware IP. *Proceedings of the 2015 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2015*, pages 163–168. URL: https://ieeexplore.ieee.org/abstract/document/7140256 Accessed March 30, 2022.

Blech, J. O. and Biha, S. O. (2013). On Formal Reasoning on the Semantics of PLC Using Coq. *arXiv preprint arXiv:1301.3047*. URL: https://arxiv.org/abs/1301.3047 Accessed March 30, 2022.

Bluespec Inc. (2012a). *Bluespec™SystemVerilog Reference Guide*. URL: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf Accessed March 29, 2022.

Bluespec Inc. (2012b). Learning Bluespec. URL: http://wiki.bluespec.com/Home Accessed March 29, 2022.

Bluespec Inc. (2019). Open Source RISC-V Cores and Tools. Bluespec Inc. URL: https://bluespec.com/, Accessed March 29, 2022.

Bluespec Inc. (2020). Bluespec, Inc. to Open Source Its Proven BSV High-level HDL Tools — Bluespec. https://bluespec.com/2020/01/06/bluespec-inc-to-open-source-its-proven-bsv-high-level-hdl-tools/.

Bluespec Inc. (2021). B-Lang-org/bsc: Bluespec Compiler (BSC). URL: https://github.com/B-Lang-org/bsc.git Accessed March 29, 2022.

Bonfanti, S., Gargantini, A., and Mashkoor, A. (2020). Design and validation of a C++ code generator from abstract state machines specifications. *Journal of Software: Evolution and Process*, 32(2):e2205. URL: https://cs.unibg.it/gargantini/research/papers/asm2Cpp_JSEP.pdf Accessed August 30, 2022.

Bourgeat, T., Pit-Claudel, C., and Chlipala, A. (2020). The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257. ACM New York, NY, USA. URL: http://adam.chlipala.net/papers/KoikaPLDI20/KoikaPLDI20.pdf Accessed August 30, 2022.

Bowen, K. A. (1979). *Model Theory for Modal Logic — Kripke Models for Modal Predicate Calculi*, volume 127 of *Studies in Epistemology, Logic, Methodology, and Philosophy of Science*. D. Reidel Publishing Company. URL: https://books.google.ca/books?id=VBPvCAAAQBAJ Accessed March 30, 2022.

Braibant, T. and Chlipala, A. (2013). Formal Verification of Hardware Synthesis. In *Computer Aided Verification*, volume 8044, pages 213–228. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-642-39799-8_14 Accessed March 30, 2022.

Brandt, J., Schneider, K., and Shukla, S. (2010). Translating Concurrent Action Oriented Specifications to Synchronous Guarded Actions. *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers,* *and Tools for Embedded Systems (LCTES)*, pages 47–56. URL: `https://dl.acm.org/doi/abs/10.1145/1755951.1755896` Accessed March 30, 2022.

Brummayer, R. and Biere, A. (2009). Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer. `https://link.springer.com/chapter/10.1007/978-3-642-00768-2_16` Accessed March 30, 2022.

Burlyaev, D. (2015). *Design, optimization, and formal verification of circuit fault-tolerance techniques*. PhD thesis, Université Grenoble Alpes. URL: `https://tel.archives-ouvertes.fr/tel-01253368/` Accessed March 30, 2022.

Camilleri, A., Gordon, M., and Melham, T. F. (1986). *Hardware Verification Using Higher-Order Logic*. University of Cambridge, Computer Laboratory. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-91.html` Accessed March 30, 2022.

Catano, Néstor and Rivera, Víctor (2016). EventB2Java: A code generator for Event-B. In *NASA Formal Methods Symposium*, pages 166–171. Springer. URL: `https://dl.acm.org/doi/abs/10.1007/978-3-319-40648-0_13` Accessed August 30, 2022.

Chen, G. (2012). A Short Historical Survey of Functional Hardware Languages. *International Scholarly Research Notices*, 2012. URL: `https://downloads.hindawi.com/archive/2012/271836.pdf` Accessed March 30, 2022.

Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., et al. (2017). Kami: A Platform for High-Level Parametric Hardware Specification and its Modular Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):24. URL: `https://dspace.mit.edu/handle/1721.1/134865` Accessed March 30, 2022.

Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. (2013). The MathSAT5 SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages

93–107. Springer. https://link.springer.com/chapter/10.1007/978-3-642-36742-7_7 Accessed March 30, 2022.

Davis, J. and Reese, R. (2008). *Finite State Machine Datapath Design, Optimization, and Implementation.* Morgan & Claypool. URL: https://www-morganclaypool-com.libaccess.lib.mcmaster.ca/doi/pdf/10.2200/S00087ED1V01Y200702DCS014 Accessed April 10, 2022.

Daylight, E. G. and Shukla, S. K. (2009). On the Difficulties of Concurrent-System Design, Illustrated With a 2x2 Switch Case Study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5850 LNCS:273–288. URL: https://link.springer.com/chapter/10.1007/978-3-642-05089-3_18 Accessed March 30, 2022.

De Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-540-78800-3_24 Accessed March 30, 2022.

Durand, S. H. and Bonato, V. (2012). A Tool to Support Bluespec SystemVerilog Coding Based on UML Diagrams. In *IECon 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 4670–4675. IEEE. URL: https://ieeexplore.ieee.org/abstract/document/6389493 Accessed March 30, 2022.

Dutertre, B. and De Moura, L. (2006). The Yices SMT Solver. URL: http://yices.csl.sri.com/tool-paper.pdf Accessed March 29, 2022.

Erkök, L. (2019). SBV: SMT Based Verification: Symbolic Haskell Theorem Prover Using SMT Solving. https://hackage.haskell.org/package/sbv Accessed March 29, 2022.

Erkök, L. (2020). sbv :: Stackage Server. URL: https://www.stackage.org/lts-14.23/package/sbv-8.3 Accessed March 29, 2022.

Fitzgerald, J., Bicarregui, J., Larsen, P. G., and Woodcock, J. (2013). Industrial Deployment of Formal Methods: Trends and Challenges.

In *Industrial Deployment of System Engineering Methods*, pages 123–143. Springer. URL: `https://link.springer.com/chapter/10.1007/978-3-642-33170-1_10` Accessed March 30, 2022.

Forsyth, M., editor (2014). *"Theorem", Collins English Dictionary — Complete and Unabridged.* HarperCollins Publishers, 12th edition. URL: `https://www.collinsdictionary.com/dictionary/english/theorem` Accessed March 30, 2022.

Gala, N., Menon, A., Bodduna, R., Madhusudan, G., and Kamakoti, V. (2016). SHAKTI Processors: An Open-Source Hardware Initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 7–8. IEEE. URL: `https://www.computer.org/csdl/proceedings-article/vlsid/2016/8700a007/12OmNAo45Ec` Accessed March 30, 2022.

George, P., Sahoo, A., Menon, A., and Kamakoti, V. (2018). SHAKTI: An Open-Source Processor Ecosystem. *Advanced Computing and Communications.* URL: `https://doi.org/10.34048/2018.3.f2` Accessed March 30, 2022.

Gill, A. (2020). mtl :: Stackage Server. URL: `https://www.stackage.org/lts-14.23/package/mtl-2.2.2` Accessed March 29, 2022.

Gordon, M. J. and Melham, T. F. (1993). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press. URL: `https://dl.acm.org/doi/abs/10.5555/155278` Accessed August 11, 2022.

Hall, A. (1990). Seven Myths of Formal Methods. *IEEE software*, 7(5):11–19. URL: `https://doi.org/10.1109/52.57887` Accessed March 30, 2022.

Hudak, P. and Fasel, J. H. (1992). A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52. URL: `https://dl.acm.org/doi/pdf/10.1145/130697.130698` Accessed August 11, 2022.

IEC (2013). *61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages.* International Electrotechnical Commission. Proprietary Standard, no URL available.

Isen, C., John, L. K., and John, E. (2009). A Tale of Two Processors: Revisiting the RISC-CISC Debate. In *Spec benchmark workshop*, pages 57–76. Springer. URL: `https://link.springer.com/chapter/10.1007/978-3-540-93799-9_4` Accessed March 30, 2022.

Jensen, K. and Wirth, N. (2012). *PASCAL User Manual and Report: ISO PASCAL Standard*. Springer Science & Business Media. URL: `https://books.google.ca/books?id=NvHjBwAAQBAJ` Accessed March 30, 2022.

Kleinedler, S. R., editor (2016). *"Theorem", American Heritage Dictionary of the English Language*. Houghton Mifflin Harcourt Publishing, 5th edition. URL: `https://www.ahdictionary.com/word/search.html?q=theorem` Accessed March 30, 2022.

Leijen, D., Martini, P., and Latter, A. (2020). parsec :: Stackage Server. URL: `https://www.stackage.org/lts-14.23/package/parsec-3.1.14.0` Accessed March 29, 2022.

Leijen, D. and Meijer, E. (2001). Parsec: Direct Style Monadic Parser Combinators For The Real world.

Lööw, A. (2021). Lutsig: a verified Verilog compiler for verified circuit development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 46–60. URL: `https://www.doc.ic.ac.uk/~aloow/papers/cpp2021.pdf` Accessed August 30, 2022.

Lööw, Andreas and Kumar, Ramana and Tan, Yong Kiam and Myreen, Magnus O and Norrish, Michael and Abrahamsson, Oskar and Fox, Anthony (2019). Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1041–1053. URL: `https://cakeml.org/pldi19.pdf` Accessed August 30, 2022.

Lööw, Andreas and Myreen, Magnus O (2019). A proof-producing translator for Verilog development in HOL. In *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 99–108. IEEE. URL: `https://ieeexplore.ieee.org/abstract/document/8807452` Accessed August 30, 2022.

Lorenz, E. N. (1995). *The Essence of Chaos*. University of Washington Press. URL: https://www.google.ca/books/edition/The_Essence_Of_Chaos/CGm2IEWH894C Accessed March 30, 2022.

Madhusudan, G. S. (2018). casl / rapidio / old_src / Logical_Transport / BSV - Bitbucket. URL: https://bitbucket.org/casl/rapidio/src/master/old_src/Logical_Transport/BSV/ Accessed March 29, 2022.

Marlow, S. (2010). Happy: The Parser Generator for Haskell. URL: http://www.haskell.org/happy/ Accessed March 29, 2022.

Menon, A., Murugan, S., Rebeiro, C., Gala, N., and Veezhinathan, K. (2017). Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, page 2. ACM. URL: https://dl.acm.org/doi/abs/10.1145/3092627.3092629 Accessed March 30, 2022.

Miller, S. P. and Srivas, M. (1995). Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *Industrial-Strength Formal Specification Techniques, 1995. Proceedings., Workshop on*, pages 2–16. IEEE. URL: https://ieeexplore.ieee.org/abstract/document/515475 Accessed March 30, 2022.

Mitchell, N. (2020). extra :: Stackage Server. URL: https://www.stackage.org/lts-14.23/package/extra-1.6.18 Accessed March 29, 2022.

Moore, N. (2022). BAPIP Project Homepage. URL: https://github.com/nmoore771/bapip Accessed March 29, 2022.

Moore, N. and Lawford, M. (2017). Correct Safety Critical Hardware Descriptions via Static Analysis and Theorem Proving. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 58–64. IEEE. URL: https://ieeexplore.ieee.org/abstract/document/7967994 Accessed March 30, 2022.

Morin-Allory, K. and Borrione, D. (2006). Automatic Generation of a Provable Circuit Model: from VHDL to PVS. In *8th International Mathematica Symposium*. URL: https://hal.univ-grenoble-alpes.fr/hal-00142692/ Accessed March 30, 2022.

Newell, J., Pang, L., Tremaine, D., Wassyng, A., and Lawford, M. (2016). Formal Translation of IEC 61131-3 Function Block Diagrams to PVS with Nuclear Application. pages 206–220. URL: https://link.springer.com/chapter/10.1007/978-3-319-40648-0_16 Accessed March 30, 2022.

Newell, J., Pang, L., Tremaine, D., Wassyng, A., and Lawford, M. (2018). Translation of IEC 61131-3 Function Block Diagrams to PVS for Formal Verification with Real-Time Nuclear Application. *Journal of Automated Reasoning*, 60(1):63–84. URL: https://link.springer.com/article/10.1007/s10817-017-9415-7 Accessed March 30, 2022.

Nguyen, T. (2011). What is the World's Data Storage Capacity? URL: https://www.zdnet.com/article/what-is-the-worlds-data-storage-capacity/ Accessed March 29, 2022.

Nikhil, R. (2004). Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004.*, pages 69–70. IEEE. URL: https://ieeexplore.ieee.org/abstract/document/1459818 Accessed March 30, 2022.

Nikhil, R. S. (2008). Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In *High-Level Synthesis*, pages 129–146. Springer. URL: https://link.springer.com/chapter/10.1007/978-1-4020-8588-8_8 Accessed March 30, 2022.

Ostroumov, S., Tsiopoulos, L., Sere, K., and Plosila, J. (2013). Generation of Structural VHDL Code with Library Components from Formal Event-B Models. *Proceedings - 16th Euromicro Conference on Digital System Design, DSD 2013*, pages 111–118. URL: https://ieeexplore.ieee.org/abstract/document/6628267 Accessed March 30, 2022.

Ostroumov, S. and Waldén, M. (2015). Formal Library of Visual Components. Technical report, TUCS Technical Report 1147, Turku Centre for Computer Science, Turku. URL: https://www.researchgate.net/profile/Sergey-Ostroumov/publication/297148975_Formal_Library_of_Visual_Components Accessed March 30, 2022.

O'Sullivan, B. (2020). text :: Stackage Server. URL: `https://www.stackage.org/package/text` Accessed March 29, 2022.

Ouchet, F., Borrione, D., Morin-Allory, K., and Pierre, L. (2009). High-Level Symbolic Simulation for Automatic Model Extraction. In *12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 218–221. IEEE. URL: `https://ieeexplore.ieee.org/abstract/document/5012132` Accessed March 30, 2022.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A Prototype Verification System. In *Automated Deduction-CADE-11*, pages 748–752. Springer. URL: `https://link.springer.com/content/pdf/10.1007/3-540-55602-8_217.pdf` Accessed March 30, 2022.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. (2001). *PVS Language Reference*. URL: `https://pvs.csl.sri.com/doc/pvs-language-reference.pdf` Accessed March 30, 2022.

Pang, L., Wang, C.-W., Lawford, M., and Wassyng, A. (2013). Formalizing and Verifying Function Blocks Using Tabular Expressions and PVS. In *Formal Techniques for Safety-Critical Systems*, volume 419, pages 125–141. Springer. URL: `https://link.springer.com/chapter/10.1007/978-3-319-05416-2_9` Accessed March 30, 2022.

Pang, L., Wang, C. W., Lawford, M., and Wassyng, A. (2015). Formal Verification of Function Blocks Applied to IEC 61131-3. *Science of Computer Programming*, 113:149–190. URL: `https://www.sciencedirect.com/science/article/pii/S0167642315002981` Accessed March 30, 2022.

Pizani Flor, J. P. (2014). Π-Ware: An Embedded Hardware Description Language using Dependent Types. URL: `https://archive.alvb.in/msc/thesis/repo-archive/thesis/main.pdf` Accessed March 30, 2022.

Porter III, H. H. (2018). RISC-V: An Overview of the Instruction Set Architecture. URL: `https://web.cecs.pdx.edu/~harry/riscv/RISCV-Summary.pdf` Accessed March 29, 2022.

RapidIO.org (2017). *RapidIO TM Interconnect Specification — Part 1: Input/Output Logical Specification*, 4.1 edition. URL: `https://rapidio.org/files/IO_logical.pdf` Accessed March 30, 2022.

Rashid, M., Waseem, M., and Khan, A. M. (2015). Toward the Tools Selection in Model Based System Engineering for Embedded Systems — A Systematic Literature Review. *The Journal of Systems & Software*, 106:150–163. URL: `https://www.sciencedirect.com/science/article/abs/pii/S016412121500103X` Accessed March 30, 2022.

Richards, D. (2011a). Automated Reasoning for Bluespec Designs. URL: `https://sourceforge.net/projects/ar4bluespec/files/` Accessed April 15, 2022.

Richards, D. (2011b). *Hardware languages and proof.* The University of Manchester (United Kingdom). URL: `http://apt.cs.manchester.ac.uk/ftp/pub/amulet/OLD_theses/D_Richards11_phd.pdf` Accessed August 11, 2022.

Richards, D. and Lester, D. (2011). A Monadic Approach to Automated Reasoning for Bluespec SystemVerilog. *Innov. Syst. Softw. Eng.*, 7(2):85–95. URL: `https://link.springer.com/article/10.1007/s11334-011-0149-0` Accessed March 30, 2022.

RISC-V Foundation (2020). RISC-V Foundation — Instruction Set Architecture (ISA). URL: `https://riscv.org/` Accessed March 29, 2022.

Rushby, J., Owre, S., and Shankar, N. (1998). Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709. URL: `https://ieeexplore.ieee.org/abstract/document/713327` Accessed March 30, 2022.

Saeed, N., Inam, A., Khan, A., and Hasan, O. (2012). V-HOLT Verifier — An Automatic Formal Verification Tool For Combinational Circuits. *2012 15th International Multitopic Conference, INMIC 2012*, pages 0–3. URL: `https://ieeexplore.ieee.org/abstract/document/6511465` Accessed March 30, 2022.

Stappers, F. P. M., Reniers, M. A., and Groote, J. F. (2010). Suitability of mCRL2 for Concurrent-System Design: A 2 x 2 Switch Case Study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6286 LNCS:166–185. URL: `https://link.springer.com/chapter/10.1007/978-3-642-17071-3_9` Accessed March 30, 2022.

Terei, D. (2020). pretty :: Stackage Server. URL: `https://www.stackage.org/lts-14.23/package/pretty-1.1.3.6` Accessed March 29, 2022.

Uma, V., Marimuthu, R., and Hicks, J. (2022). Formal verification of a 4 bit counter using kami verification flow. In *AIP Conference Proceedings*, volume 2393, page 020081. AIP Publishing LLC. URL: `https://aip.scitation.org/doi/abs/10.1063/5.0074153` Accessed August 30, 2022.

Vijayaraghavan, M., Chlipala, A., Dave, N., et al. (2015). Modular Deductive Verification of Multiprocessor Hardware Designs. In *International Conference on Computer Aided Verification*, pages 109–127. Springer. URL: `https://link.springer.com/chapter/10.1007/978-3-319-21668-3_7` Accessed March 30, 2022.

Wassyng, A., Lawford, M., and Hu, X. (2005). Timing Tolerances in Safety-Critical Software. In *International Symposium on Formal Methods*, pages 157–172. Springer. URL: `https://link.springer.com/chapter/10.1007/11526841_12` Accessed March 30, 2022.

Wassyng, A., Lawford, M. S., and Maibaum, T. S. (2011). Software Certification Experience in the Canadian Nuclear Industry: Lessons for the Future. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 219–226. ACM. URL: `https://dl.acm.org/doi/abs/10.1145/2038642.2038676` Accessed March 30, 2022.

Waterman, A. and Asanović, K. (2017). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2.2 edition. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf` Accessed March 30, 2022.

Wenzel, M., Paulson, L. C., and Nipkow, T. (2008). The isabelle framework. In *International Conference on Theorem Proving in Higher Order Logics*, pages 33–38. Springer. URL: `https://link.springer.com/chapter/10.1007/978-3-540-71067-7_7` Accessed August 11, 2022.

Yorgey, B. (2020). split :: Stackage Server. URL: `https://www.stackage.org/lts-14.23/package/split-0.2.3.3` Accessed March 29, 2022.

APPENDIX

A. BAPIP: A BSV-TO-PVS TRANSLATOR

The semantic translation of BSV into PVS presented in §4 and §3 has been algorithmically encoded in the BAPIP translation tool. It features a simple, command-line interface, fast execution times, and an easily extensible modular structure.

A.1 Translator Architecture

An overview of the translator's architecture in flow-chart form is presented in Figure A.1. The path through this diagram selects which of the four translation modes is selected: BSV2PVS, BSV2BSV, TSP2BSV, and TSP2PVS. In this diagram, ellipses indicate data in concrete form, such as a file or a populated data structure, whereas the rectangles are the modularized transformation algorithms which translate between representations. While Figure A.1 may give the appearance of a branching structure, it is more accurate to consider this architecture a collection of independent data pipelines which share components. For example, in BSV2PVS mode, the translator has no interaction whatsoever with either the BSV generator or the Tabular Specification parser. Not depicted is the top-level control, which handles system I/O, data passing, the invocation of algorithms, and other functions.

While it may seem somewhat vacuous to have a BSV2BSV mode, it divulges interesting information in addition to being very simple to implement after having implemented TSP2BSV. One use is to re-organize BSV files. Comparing a file before and after parsing and generation demonstrates what information (if any) is discarded as superfluous. Additionally, the BSV parser features permutation parsing, but does not preserve the order in which elements occurred. Rather, it groups like elements and encodes them as abstract syntax. It is therefore possible to automatically organize code via this method. Such modes are also useful during the extension of BAPIP to other languages, as it provides a method for testing both parser and generator before the completion of the translation algorithms.

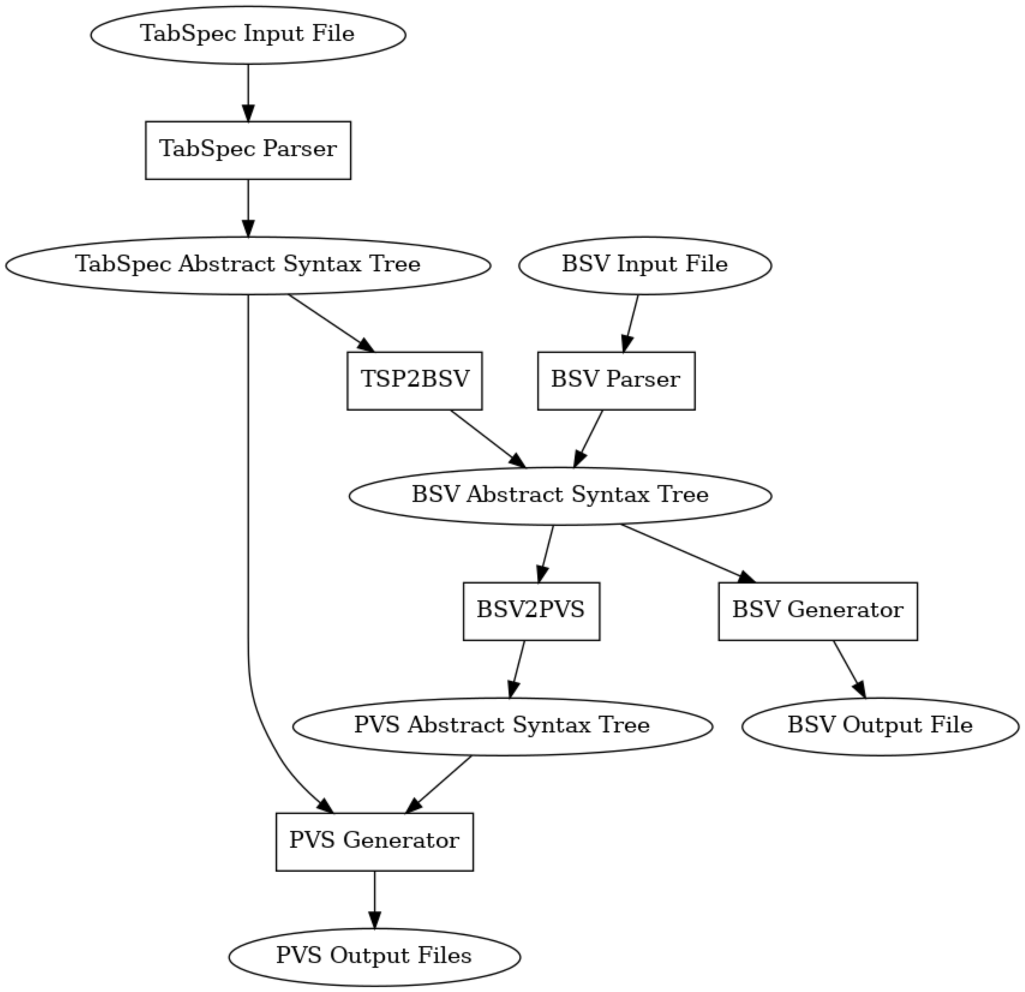


Figure A.1: Expanded BAPIP Architectural Overview

It should be noted that, in TSP2PVS mode, the original tabular specifications used to generate BSV code are required by the PVS generation algorithm, in order to generate PVS files with working proofs of correctness. As such, the PVS generator requires access to the TSP abstract syntax tree, if such is available. If no TSP abstract syntax is available, such as when the translator is invoked in BSV2PVS mode, no tabular specifications or proofs of correctness are generated in the resulting PVS files.

A.1.1 Organization of PVS Output

As the result of the PVS Generator, the translator produces four PVS theories, in separate files, which are written into the selected output folder. Files are listed in the order of their package hierarchy.

`TypeDefinitions.pvs`	Basic type definitions; translations of all type definitions, constant declarations, enumeration declarations; type initializations for record types; functions
`State.pvs`	State record types for all instantiated modules; state record instantiation predicates
`Methods.pvs`	Definitions for all value methods declared in the bsv module hierarchy
`Transitions.pvs`	Schedule indexed transition predicates, transition functions returning updated state records
`MyModule.pvs`	Theorem template; transition schedule documentation; auto-generated consistency theorems and proofs

Tab. A.1: Output files produced by BAPIP

Basic type definitions described in `TypeDefinitions.pvs` provide analogs for BSV's `Int`, `UInt` and `Bit` types. These types take a bit-width as an argument, and in PVS these translate to sub-ranged integers.

BAPIP is designed to minimize the number of libraries the user must install in PVS prior to the use of BAPIP produced PVS files. As such, the files in Table A.2 are automatically generated by BAPIP and placed into the specified output directory as part of any process producing PVS output.

A.2 Prerequisites

BAPIP has been designed as a stand-alone executable, and requires no preinstalled software to execute. However, in order to use the BAPIP translation tool for its intended purpose, certain prerequisite software must be installed. This section presents a comprehensive list of all software dependencies.

`arith_bitwise.pvs`	defines bitwise operators and operations as equivalent arithmetic operations.
`ClockTick.pvs`	library by (Wassyng et al., 2005). Defines the `tick` type.
`defined_operators.pvs`	library by (Wassyng et al., 2005). Included mainly to maintain compatibility with Pang's tabular specifications
`FIFO.pvs`	defines a record type and associated functions implementing First-In-First-Out Buffer (FIFO) buffers, and the common access functions specified in the BSV documentation.
`Maybe.pvs`	defines a record type and associated functions implementing a Maybe-like type. A detailed examination of `Maybe` may be found in §3.
`Time.pvs`	library by (Wassyng et al., 2005). Supports `ClockTick.pvs`

Tab. A.2: Associated PVS libraries automatically generated

In addition to the requirements below, a computer with a Linux/Unix based operating system is required in order to use the BAPIP tool chain. The executable which is available at (Moore, 2022) has been compiled under Ubuntu 18, and as such should be executable on most Debian-family operating systems. The source code is also available from the same website for recompilation.

A.2.1 Installation of Bluespec SystemVerilog

For the overwhelming majority of the work here under examination, the Bluespec compiler was a proprietary, controlled-source tool, made available to the author through Bluespec Inc.'s university program. On Jan 6, 2020, Bluespec Inc. announced the release of their compiler under an open-source framework. The latest version, 2021.07, was released in July of 2021 at (Bluespec Inc., 2021).

Installation of BSV is not strictly necessary for the operation of BAPIP, however, all files translated by BAPIP are assumed to have passed typechecking by BSC (the Bluespec Compiler) prior to translation. This is not because BAPIP requires a post-compilation file of any kind, but because it allows the translation to make certain assumptions about the code being entered into it. As the project has worn on and the translator made more robust, less of these assumptions have become necessary, but it is still assumed (for example) that the BSV design adheres to BSV's restrictions on valid identifier characters.

A.2.2 Installation and Configuration of PVS

The latest PVS release, version 7.1, was released in May of 2020. The BAPIP package, as well as associated case studies and proofs will not work with the new version of PVS, so use of PVS 6.0 with the BAPIP system is required, for the following reasons.

The PVS project maintains a github repository where the latest updates can be found. On the github page, PVS appears to have been renamed from "Prototype Verification System" to "The People's Verification System." As with all polities throughout history which have adopted communism, PVS's latest versions fail to work as stated or intended. While case study 4 can be loaded into PVS 7.1, re-entering the proof which works in PVS 6.0 results in an "undefined function" error of mysterious origin and no known means of escape. As such, PVS 6.0 is recommended.

PVS 6.0 must use one of two Lisp back-ends to operate, Allegro or SBCL. The PVS binaries available from the PVS homepage can optionally include a distribution of the proprietary Allegro Lisp compiler. Allegro is proprietary and must be bought to be used. Although there is a free version, the licensing agreement for it states that the use of the free version of Allegro specifically for the purposes of University sanctioned academic research is prohibited. The fee associated with use of Allegro is sizable enough for the author to recommend installation of SBCL, though this somewhat complicates installation.

The most recent stable version of SBCL with which PVS 6.0 is compatible is SBCL 1.4.14. Once installed, PVS requires the manual creation of a environment variable pointing to the directory of the Lisp executable. Once this is configured, PVS may be compiled and installed.

However, the use of SBCL introduces an interesting problem. For theorem proving at the size and scale necessary for the case study in §6.5, it is

package name	version requirement
base	>= 4.7 and < 5
text (O'Sullivan, 2020)	none
parsec (Leijen et al., 2020)	none
directory (hkg, 2020b)	none
pretty (Terei, 2020)	none
sbv (Erkök, 2020)	none
mtl (Gill, 2020)	none
containers (hkg, 2020a)	none
extra (Mitchell, 2020)	none
split (Yorgey, 2020)	none

Tab. A.3: List of Haskell Library Dependencies

necessary to increase the default 400 MB of garbage collection space allocated to SBCL. Since SBCL is invoked automatically by the PVS startup routines via Emacs lisp, it was necessary in the author's case to modify one of the PVS Emacs lisp files to invoke this added command-line flag. Specifically, on lines 145 and 146 of the file `emacs/pvs-ilisp.el`, the lisp expression `(defun pvs-program () pvs-image)` was replaced with `(defun pvs-program () (format "%s --dynamic-space-size 2048" pvs-image))`. This increases the default allocation of 400 MB to 2GB, which proved sufficient for the progress towards our correctness conditions represented in §6.5.

A.2.3 Compile-time Requirements

If a specific distribution of Linux cannot execute the program provided at (Moore, 2022), it is recommended that the program be recompiled from source. In order to do so, one must minimally have the Glorious Glasgow Haskell Compiler (GHC), version 8.0.2. Recompiling using stack will automatically fetch the libraries specified in `package.yaml`. Those specific dependencies are listed in Table A.3, along with links to the associated library homepages.

Parsec (Leijen and Meijer, 2001) was selected over Happy (Marlow, 2010) as a parsing library for this project for a number of reasons. As a top-down parser, Parsec is more natural for parsing BSV files, which are reasonably structured from a top-down perspective. Parsec also integrates the full expressive power of Haskell, as opposed to Happy, which is a domain specific language.

A.3 Unmodelled Behaviours

While an exhaustive translation from BSV to PVS would be a grand contribution to the field of formal methods, this is unfortunately not possible given the resources allocated to this project. As such, the language elements included in the translation have been triaged so that the most essential functions have been added first. The following list of unmodelled language elements are considered future work for the BAPIP translation project.

A.3.1 Other Pragmas

Besides those controlling action arbitration, there are many different pragmas. While many of these are useful for development in Bluespec, the integration of them with the existing translator structure is reserved as future work.

A.3.2 Type Classes and Type Class Declarations

A potentially useful future addition to the translator would be the semantic interpretation of the type classes included in type definitions. These could be used to impose/remove additional constraints from custom types, improving the subset of Bluespec SystemVerilog covered by the translator. Additionally, user-generated type-classes are not addressed.

A.3.3 Parameterized Modules

One of the powerful abstraction mechanisms available in BSV is the ability to create parameterized modules. This capability is not currently supported. Supporting it would require a reasonably drastic re-organization of the post-translation PVS semantics. Currently, the modularity of Bluespec modules is preserved only insofar as they extend to the state record. The implementation of this feature is reserved as future work.

A.4 Operational Instructions

The BAPIP translation tool is simple and easy to use. To use it, one must invoke it from the command line using `stack exec` and specify a mode: `bsv2pvs`, `bsv2bsv`, `tsp2bsv`, or `tsp2pvs`. Modes using BSV and Tabular Specifications as source languages are viable, but the implementation of modes with PVS as the source language are reserved as future work. The `help` command may also be used to display usage information.

It is then necessary to specify a BSV package as the source file. It is important to note that this source package's dependencies must exist in the same directory as the source file, unless it is part of Bluespec's library packages, otherwise the package will not be found by the translator. Next, it is necessary to specify which module within the selected package is the top-level module under examination. This information is necessary for the production of transition predicates, and cannot be deduced by the algorithm. If no valid module is selected, the translator fails execution, providing a list of modules which exist within the selected package. As a final argument, the user may optionally specify an output directory for the translator to put the translated files. In cases where the required module information is omitted, the software will display a list of applicable module names. Some packages do not contain modules, such as those containing only type definitions or function declarations. In such cases, the module name *may* be omitted correctly.

The translator will always create a new directory to place its output. If the translator is instructed to place the output in a certain directory and that directory already exists, it will create a new directory, suffixed with "`_VXXX`", where `XXX` is the number of pre-existing suffixed directories incremented by one. It is therefore possible to store 1000 versions of the same translated module in the same directory. If the translator receives no indication of an output directory, it will put the output directory in the translator's own home directory, named with the top level module with the suffix "`-pvs`," and optionally the version number suffix.

Once invoked, the BAPIP translation tool requires no further input from the user, and will either successfully execute the translation process or indicate a syntax error in the selected BSV package, or a scheduling error.

A.4.1 User Specification of Top-Level Method Invocations

While methods always take precedence over rules with respect to action arbitration, the set of methods available for scheduling must have been invoked by a supermodule during that clock cycle. If we consider that each method either fires or does not fire, each set of methods creates a particular and possibly unique universal schedule. For clarity, we will refer to these as "plans".

The cardinality of the set of possible plans is 2^n, where n is the number of action methods in the module. For the target top-level module of the case study in §5, $n = 61$, yielding 2.305843×10^{18} possible plans. Needless to say, brute force verification over all plans is not possible at such scales. Fortunately, we are once again in a position that the overwhelming majority of these plans are not of interest for the purposes of verification.

Since the construction of a proof sequent is performed by a person rather than an algorithm, the person constructing the algorithm can use their knowledge of the system to determine which method invocations are appropriate at which times. The data-paths for such verification procedures on industrial-scale examples (such as RapidIO) indicate that a conservative estimate of the maximum number of chained transition predicates would be around 5 or 6. It is reasonable for the user to dictate method behaviour at such timescales.

By default, BAPIP generates one plan wherein no methods are called, and provides a mechanism for the user to specify custom plans. Upon execution, BAPIP will search for a `schedules.bapip` file in the same directory as the input file, and generates any plans it finds. A single plan is specified as a space-delimited list of method names. Individual plans are delimited by newlines. All plans must be declared explicitly by the user, excepting the plan wherein no input methods are invoked.

One drawback of the above approach is that the structure is somewhat rigid, and lacking some modern conveniences, such being able to generate multiple schedules from one set of methods by making some of the methods optional. Such improvements are left as future work.

A full listing of the code for this project results in the thesis document over a thousand pages. As such, these appendices are supplemented by a digital appendix, available at `https://github.com/nmoore771/bapip`. The code listing for our traffic light example (§C), and all others are reserved for the digital appendix.

B. FULL CODE LISTING FOR BAPIP TRANSLATION TOOL

The full code listing for the BAPIP translation software is available in the digital appendices at `https://github.com/nmoore771/bapip`.

B.1 Haskell Source Files

B.1.1 BAPIP.hs

B.1.2 BSV2PVS.hs

B.1.3 BSVGenerator.hs

B.1.4 BSVLexer.hs

B.1.5 ConflictSolver.hs

B.1.6 HEXLexer.hs

B.1.7 LexerTypes.hs

B.1.8 LiteralLexer.hs

B.1.9 MacroProcessor.hs

B.1.10 PVS2BSV.hs

B.1.11 PVSGenerator.hs

B.1.12 SourceFiles.hs

B.1.13 TSP2BSV.hs

B.1.14 TSPLexer.hs

C. FULL CODE LISTING FOR TRAFFIC SIGNALS EXAMPLE

What follows are full code listings for the scheduling example presented in §4.2.

C.1 BSV Source Files

C.1.1 TrafficSignals.bsv

BSV

```
package TrafficSignals;

  interface TrafficSignals ;
    method Action reset ();
    method Action pedestrian_request_NS();
    method Action pedestrian_request_EW();
    method Bit#(2) getlamp_NS();
    method Bit#(2) getlamp_EW();
    method Bool getPedestrianLamp_NS();
    method Bool getPedestrianLamp_EW();
  endinterface

  (*descending_urgency = "reset,pedestrian_request_NS,
      pedestrian_request_EW"*)
  module mkTrafficSignals (TrafficSignals);
    Reg#(Bit#(2)) carLamps_NS <- mkReg(2);
    Reg#(Bit#(2)) carLamps_EW <- mkReg(2);
    Reg#(Bit#(9)) t <- mkReg(0);

    rule tick;
      if (t < 300)
        t <= t + 1;
      else
        t <= 0;
    endrule

    rule goYellow_NS ((carLamps_NS == 2'd0) && (t == 9'd140));
      carLamps_NS <= 1;
    endrule

    rule goRed_NS ((carLamps_NS == 2'd1) && (t == 9'd160));
      carLamps_NS <= 2;
    endrule

    rule goGreen_NS ((carLamps_NS == 2'd2) && (t == 9'd0));
      carLamps_NS <= 0;
    endrule

    rule goYellow_EW ((carLamps_EW == 2'd0) && (t == 9'd0));
      carLamps_EW <= 1;
    endrule

    rule goRed_EW ((carLamps_EW == 2'd1) && (t == 9'd20));
      carLamps_EW <= 2;
    endrule

    rule goGreen_EW ((carLamps_EW == 2'd2) && (t == 9'd160));
      carLamps_EW <= 0;
    endrule

    method Action reset ();
      carLamps_NS <= 2;
      carLamps_EW <= 2;
      t <= 0;
    endmethod

    method Action pedestrian_request_NS();
      if (carLamps_EW == 0 && t < 9'd280)
        t <= 280;
    endmethod

    method Action pedestrian_request_EW();
```

```
      if (carLamps_NS == 0 && t < 9'd120)
        t <= 120;
    endmethod

    method Bit#(2) getlamp_NS();
      return carLamps_NS;
    endmethod

    method Bit#(2) getlamp_EW();
      return carLamps_EW;
    endmethod

    method Bool getPedestrianLamp_NS();
      return carLamps_NS == 0;
    endmethod

    method Bool getPedestrianLamp_EW();
      return carLamps_EW == 0;
    endmethod

  endmodule : mkTrafficSignals

endpackage : TrafficSignals
```

END BSV

C.2 Generated PVS Files

C.2.1 TypeDefinitions.pvs

PVS

```
TypeDefinitions : theory

begin

  importing arith_bitwise
  importing Maybe
  importing FIFO
  Int(n : int): TYPE = {i:int | −(2^(n−1)) <= i AND i < 2^(n−1)}
  UInt(n : int): TYPE = {i:int | 0 <= i AND i < 2^n}
  Bit(n : int): TYPE = {i:int | 0 <= i AND i < 2^n}

  mkInt(n : int)  : Int(n) = 0
  mkUInt(n : int) : UInt(n) = 0
  mkBit(n : int)  : Bit(n) = 0
















end TypeDefinitions
```

END PVS

C.2.2 State.pvs

PVS

```
State : theory

begin

  importing TypeDefinitions

   TrafficSignals  : type =
     [# t  : Bit(9)
      , carLamps_EW : Bit(2)
      , carLamps_NS : Bit(2)
     #]

  TrafficSignals_var : var TrafficSignals

  mkTrafficSignals (TrafficSignals_var) : bool
  = TrafficSignals_var‘t    = 0
  AND TrafficSignals_var‘carLamps_EW = 2
  AND TrafficSignals_var‘carLamps_NS = 2



end State
```

END PVS

C.2.3 Methods.pvs

PVS

```
Methods : theory

begin

  importing State


  getPedestrianLamp_EW (index : nat, pre : TrafficSignals, mod :
      TrafficSignals) : bool = IF (index = 0)
    THEN ( mod‘carLamps_EW = 0 )
    ELSE IF (index = 1)
    THEN ( mod‘carLamps_EW = 0 )
    ELSE False
    ENDIF ENDIF

  getPedestrianLamp_NS (index : nat, pre : TrafficSignals, mod :
      TrafficSignals) : bool = IF (index = 0)
    THEN ( mod‘carLamps_NS = 0 )
    ELSE IF (index = 1)
    THEN ( mod‘carLamps_NS = 0 )
    ELSE False
    ENDIF ENDIF
  getlamp_EW (index : nat, pre : TrafficSignals, mod : TrafficSignals) :
      Bit(2) = IF (index = 0)
    THEN mod‘carLamps_EW
    ELSE IF (index = 1)
    THEN mod‘carLamps_EW
    ELSE 0
    ENDIF ENDIF

  getlamp_NS (index : nat, pre : TrafficSignals , mod : TrafficSignals) :
      Bit(2) = IF (index = 0)
    THEN mod‘carLamps_NS
    ELSE IF (index = 1)
    THEN mod‘carLamps_NS
    ELSE 0
    ENDIF ENDIF



end Methods
```

END PVS

C.2.4 Transitions.pvs

PVS

```
Transitions : theory

begin

  importing Methods


  transition_val (index : nat, pre : TrafficSignals ) : TrafficSignals =
    IF (index = 0) THEN
    pre with
        [ t := if ( pre‘t < 300 )
        then ( pre‘t + 1 )
        else 0
      endif
```

```
        , carLamps_EW := if (( ( pre‘carLamps_EW = 2 ) AND ( pre‘t =
            160 ) ))
          then 0
          else if (( ( pre‘carLamps_EW = 1 ) AND ( pre‘t = 20 ) ))
            then 2
            else if (( ( pre‘carLamps_EW = 0 ) AND ( pre‘t = 0 ) ))
              then 1
              else pre‘carLamps_EW
            endif
          endif
        endif
        , carLamps_NS := if (( ( pre‘carLamps_NS = 2 ) AND ( pre‘t = 0 )
            ))
          then 0
          else if (( ( pre‘carLamps_NS = 1 ) AND ( pre‘t = 160 ) ))
            then 2
            else if (( ( pre‘carLamps_NS = 0 ) AND ( pre‘t = 140 ) ))
              then 1
              else pre‘carLamps_NS
            endif
          endif
        endif
        ]
    ELSE pre
    ENDIF

  transition_val (index : nat, pre : TrafficSignals ) : TrafficSignals =
    IF (index = 1) THEN
    pre with
        [ t := 0
        , carLamps_EW := 2
        , carLamps_NS := 2
        ]
    ELSE pre
    ENDIF


  transition ( index : nat, pre, post : TrafficSignals ) : bool =
    post = transition_val (index, pre)
  transition ( index : nat, pre, post : TrafficSignals ) : bool =
    post = transition_val (index, pre)





end Transitions
```

END PVS

C.2.5 TrafficSignals.pvs

PVS

```
Theorems[(IMPORTING Time) delta_t:posreal] : theory

begin

  importing Transitions
  importing ClockTick[delta_t]

  t: VAR tick

  s : VAR [tick -> TrafficSignals]
  pre, post, TrafficSignals_var : VAR TrafficSignals




% The following transitions have been scheduled.
% 0  : Methods Invoked = {none}
%  : Input Args = {none}
%
% 1  : Methods Invoked = {reset, pedestrian_request_NS}
%  : Input Args = {none}
%
% The following arguments must be supplied to invoke the transition
    predicate.
```

```
%   index -> The index number corresponding to the schedule you wish to
    invoke
%   pre -> The pre-state of the transition predicate
%   post -> The post-state of the transition predicate
%   method arguments -> supply the arguments given in the above list of
    schedules, in the order they appear.
%
% For an example of how this is intended to work, take a look at the auto-
    generated consistency theorems below.

  %test1 : theorem <antecedents>
    %implies <consequents>



%|- consistency_0 : PROOF
%|- (then (skolem!)
%|-       (inst + "transition_val (i!1,pre!1)")
%|-       (rewrite transition)
%|-       (rewrite transition_val)
%|-       (assert))
%|- QED
  consistency_0 : Theorem
    FORALL (i : nat, pre : TrafficSignals) :
      EXISTS (post : TrafficSignals) :
        transition(i, pre, post)



%|- consistency_1 : PROOF
%|- (then (skolem!)
%|-       (inst + "transition_val (i!1,pre!1)")
%|-       (rewrite transition)
%|-       (rewrite transition_val)
%|-       (assert))
%|- QED
  consistency_1 : Theorem
    FORALL (i : nat, pre : TrafficSignals) :
      EXISTS (post : TrafficSignals) :
        transition(i, pre, post)
```

```
end Theorems
```

END PVS

D. FULL CODE LISTING FOR LIMITS ALARM CASE STUDY

Full code listings for the Limits Alarm case study (§6.1) may be found in the digital appendix to this thesis, at `https://github.com/nmoore771/bapip`.

D.1 BSV Source Files

D.1.1 LIMITS_ALARM.bsv

D.1.2 HYSTERESIS.bsv

D.2 Generated PVS Files

D.2.1 TypeDefinitions.pvs

D.2.2 State.pvs

D.2.3 Transitions.pvs

D.2.4 LIMITS_ALARM.pvs

D.3 PVS Proof File

D.3.1 LIMITS_ALARM.prf

E. FULL CODE LISTING FOR ALARM_INT CASE STUDY

Full code listings for the ALRM_INT case study (§6.3) may be found in the digital appendix to this thesis, at `https://github.com/nmoore771/bapip`.

E.1 TSP Source File

E.1.1 ALRM_INT.tsp

E.2 Generated BSV File

E.2.1 Alrm_int.bsv

E.3 Generated PVS Files

E.3.1 TypeDefinitions.pvs

E.3.2 State.pvs

E.3.3 Methods.pvs

E.3.4 Transitions.pvs

E.3.5 mkAlrm_int.pvs

F. FULL CODE LISTING FOR RAPIDIO DECODER CASE STUDY

Full code listings for the RapidIO read/write size and word pointer decoder module case study (§6.3) may be found in the digital appendix to this thesis, at `https://github.com/nmoore771/bapip`.

F.1 BSV Source Files

F.1.1 RapidIO.defines

F.1.2 RapidIO_DTypes.bsv

F.1.3 RapidIO_RegisterFile_Offset.defines

F.1.4 RapidIO_TgtDecoder_ByteCnt_ByteEn.bsv

F.2 Generated PVS Files

F.2.1 TypeDefinitions.pvs

F.2.2 RapidIO_TgtDecoder_ByteCnt_ByteEn.pvs

F.3 PVS Proof File

F.3.1 RapidIO_TgtDecoder_ByteCnt_ByteEn.prf

G. FULL CODE LISTING FOR RAPIDIO ENCODER CASE STUDY

Full code listings for the RapidIO read/write size and word pointer encoder module case study (§6.4) may be found in the digital appendix to this thesis, at `https://github.com/nmoore771/bapip`.

G.1 BSV Source Files

G.1.1 RapidIO.defines

G.1.2 RapidIO_DTypes.bsv

G.1.3 RapidIO_InitEncoder_WdPtr_Size.bsv

G.2 Generated PVS Files

G.2.1 TypeDefinitions.pvs

G.2.2 State.pvs

G.2.3 Methods.pvs

G.2.4 Transitions.pvs

G.2.5 RapidIO_InitEncoder_WdPtr_Size.pvs

G.3 PVS Proof File

G.3.1 RapidIO_InitEncoder_WdPtr_Size.prf

H. FULL CODE LISTING FOR RAPIDIO TRANSACTION ID CASE STUDY

Full code listings for the RapidIO transaction ID echoing case study (§6.5) may be found in the digital appendix to this thesis, at `https://github.com/nmoore771/bapip`.

H.1 BSV Source Files

H.1.1 RapidIO_InComingPkt_Separation.bsv

H.1.2 RapidIO_IOPkt_Concatenation.bsv

H.1.3 RapidIO_IOPkt_Generation.bsv

H.1.4 RapidIO_MainCore.bsv

H.1.5 RapidIO_PktTransportParse.bsv

H.1.6 RapidIO_RxPktFTypeAnalyse.bsv

H.1.7 RapidIO_TargetReqIFC.bsv

H.1.8 RapidIO_TargetRespIFC.bsv

I. INCLUDED PVS LIBRARY FILES

The following are full code listings for PVS library files required for theorem proving. This standard set of libraries is included by default during every successful BAPIP translation run. Full code listings are available from `https://github.com/nmoore771/bapip`.

I.1 PVS Files

I.1.1 ClockTick.pvs

I.1.2 defined_operators.pvs

I.1.3 monad.pvs

I.1.4 Time.pvs

I.1.5 arith_bitwise.pvs